| Document Title | Specification of NVRAM Manager |
|---|---|
| **Document Owner** | AUTOSAR |
| **Document Responsibility** | AUTOSAR |
| **Document Identification No** | 033 |
| **Document Classification** | Standard |

| | |
|---|---|
| **Document Version** | 2.6.0 |
| **Document Status** | Final |
| **Part of Release** | 3.2 |
| **Revision** | 3 |

**Document Change History**

| Date | Version | Changed by | Change Description |
|---|---|---|---|
| 28.02.2014 | 2.6.0 | AUTOSAR Release Management | • Added header include parameter used for the RAM and ROM declarations<br>• Clarified explicit synchronization for multi block operations<br>• Added NvMMainFunctionPeriod parameter |
| 17.05.2012 | 2.5.0 | AUTOSAR Administration | • Support of NV data interfaces (explicit synchronization)<br>• Improved handling of redundant NVRAM block<br>• Changed behaviour of restore block defaults (init callback) |
| 07.04.2011 | 2.4.0 | AUTOSAR Administration | • Location of CRC in RAM<br>• DEM event status reporting specified<br>• Behavior specified when NVRAM block ID 1 shall be written<br>• Handling of single-block callbacks during asynchronous multi-block specified<br>• Behavior specified to prevent possible loss of data during shutdown |
| 10.09.2010 | 2.3.0 | AUTOSAR Administration | • Behavior specified to prevent possible loss of data during shutdown<br>• Typo corrected in chapter 7.1.2.1<br>• Behavior specified: handling of single-block callbacks during asynchronous multi-block requests<br>• Behavior specified when NVRAM block ID 1 shall be written<br>• Include of Crc.h is not optional<br>• Legal disclaimer revised |

| | | | |
|---|---|---|---|
| 23.06.2008 | 2.2.1 | AUTOSAR Administration | Legal disclaimer revised |
| 11.12.2007 | 2.2.0 | AUTOSAR Administration | • Technical Office SWS Improvements are incorporated.<br>• Requirement IDs for configuration parameters (chapter 10) added.<br>• Management of the RAM block state specified more precisely.<br>• The NVRAM Manager doesn't support non-sequential NVRAM block IDs any longer.<br>• Document meta information extended<br>• Small layout adaptations made |
| 26.01.2007 | 2.1.0 | AUTOSAR Administration | • AUTOSAR service description added in chapter 11<br>• Reentrancy of callback functions specified<br>• Details regarding memory hardware abstraction addressing scheme added<br>• Further changes see chapter 11<br><br>• Legal disclaimer revised<br>• “Advice for users” revised<br>• “Revision Information” added |
| 28.04.2006 | 2.0.0 | AUTOSAR Administration | Document structure adapted to common Release 2.0 SWS Template.<br>• Major changes in chapter10<br>• Structure of document changed partly<br>• Other changes see chapter 11 |
| 20.06.2005 | 1.0.0 | AUTOSAR Administration | Initial release |

## Disclaimer



## Advice for users

AUTOSAR Specification Documents may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

# Table of Contents

# 1 Introduction and functional overview

This specification describes the functionality, API and the configuration of the AUTOSAR Basic Software module NVRAM Manager.

The NvM module shall provide services to ensure the data storage and maintenance of NV data according to their individual requirements in an automotive environment. The NvM module shall be able to administrate the NV data of an EEPROM and/or a FLASH EEPROM emulation device.

The NvM module shall provide the required synchronous/asynchronous services for the management and the maintenance of NV data (init/read/write/control).

# 2 Acronyms and abbreviations

Acronyms and abbreviations, which have a local scope and therefore are not contained in the AUTOSAR glossary, must appear in a local glossary.

| ***Abbreviation/ Acronym:*** | ***Description:*** |
|---|---|
| Basic Storage Object | A “Basic Storage Object” is the smallest entity of a “NVRAM block”. Several “Basic Storage Objects” can be used to build a NVRAM Block. A “Basic Storage Object“ can reside in different memory locations (RAM/ROM/NV memory). |
| NVRAM Block | The “NVRAM Block” is the entire structure, which is needed to administrate and to store a block of NV data. |
| NV data | The data to be stored in Non-Volatile memory. |
| Block Management Type | Type of the NVRAM Block. It depends on the (configurable) individual composition of a NVRAM Block in chunks of different mandatory/optional Basic Storage Objects and the subsequent handling of this NVRAM block. |
| RAM Block | The „RAM Block“ is a „Basic Storage Object“. It represents the part of a „NVRAM Block“ which resides in the RAM.<br>See [BSW08534] .[NVM126] |
| ROM Block | The „ROM Block“ is a „Basic Storage Object“. It represents the part of a „NVRAM Block“ which resides in the ROM. The „ROM Block“ is an optional part of a „NVRAM Block“.[NVM020] |
| NV Block | The „NV Block“ is a „Basic Storage Object“. It represents the part of a „NVRAM Block“ which resides in the NV memory. The „NV Block“ is a mandatory part of a „NVRAM Block“. [NVM125] |
| Administrative Block | The “Administrative Block” is a “Basic Storage Object”. It resides in RAM. The “Administrative Block” is a mandatory part of a “NVRAM Block”. [NVM135] |
| DET | Development Error Tracer – module to which development errors are reported. |
| DEM | Diagnostic Event Manager – module to which production relevant errors are reported |
| NV | Non volatile |
| FEE | Flash EEPROM Emulation |
| EA | EEPROM Abstraction |
| FCFS | First come first served |

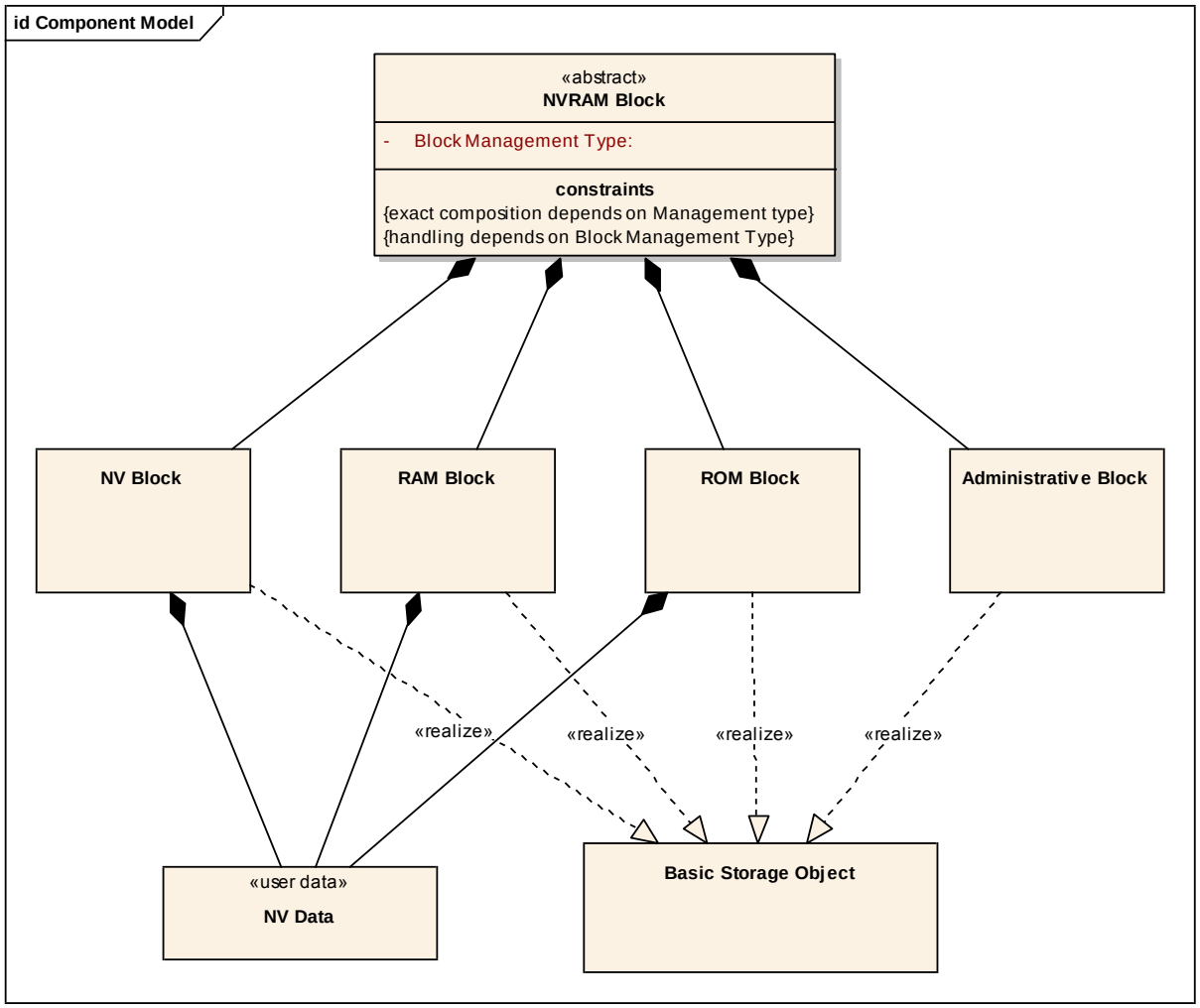


**Figure 1: Overview describing the acronyms table**

# 3 Related documentation

## 3.1 Input documents

[1] Layered Software Architecture
AUTOSAR_LayeredSoftwareArchitecture.pdf

[2] General Requirements on Basic Software Modules
AUTOSAR_SRS_General.pdf

[3] Requirements on Memory Services
AUTOSAR_SRS_MemoryServices.pdf

[4] Specification of EEPROM Abstraction
AUTOSAR_SWS_EA.pdf

[5] Specification of Flash EEPROM Emulation
AUTOSAR_SWS_FLASH_EEPROM_Emulation.pdf

[6] Specification of Memory Abstraction Interface
AUTOSAR_SWS_MemIf.pdf

[7] Specification of the Virtual Functional Bus
AUTOSAR_Spec_of_VFB.pdf

[8] Software Component Template
AUTOSAR_SoftwareComponentTemplate.pdf

[9] Specification of RTE Software
AUTOSAR_SWS_RTE.pdf

[10] Specification of BSW Scheduler
AUTOSAR_SWS_BSW_Scheduler.pdf

[11] Specification of ECU Configuration
AUTOSAR_ECU_Configuration.pdf

[12] AUTOSAR Basic Software Module Description Template,
AUTOSAR_BSW_Module_Description.pdf

# 4 Constraints and assumptions

## 4.1 Limitations

Limitations are given mainly by the finite number of “Block Management Types” and their individual treatment of NV data. These limits can be reduced by an enhanced user defined management information, which can be stored as a structured part of the real NV data. In this case the user defined management information has to be interpreted and handled by the application at least.

## 4.2 Applicability to car domains

No restrictions.

## 4.3 Conflicts

None

# 5 Dependencies to other modules

This section describes the relations to other modules within the basic software.

## 5.1 File structure

### 5.1.1 Code file structure

**NVM076**: The NvM module shall consist of the following parts:

- One or more C file NvM_xxx.c containing the entire or parts of NVRAM manager code

### 5.1.2 Header file structure

**NVM077**: The include file structure shall be as follows:

- An API interface NvM.h providing the function prototypes to access the underlying NVRAM functions
- A type header NvM_Types.h providing the types for the NvM module
- A callback interface NvM_Cbk.h providing the callback function prototypes to be used by the lower layers
- A type header NvM_Cfg.h providing the configuration parameters for the NvM module
- NvM_Cfg.h shall include NvM_Types.h
- NvM_Types.h shall include Std_Types.h
- NvM.h shall include NvM_Cfg.h
- NvM.c shall include NvM.h, Dem.h, MemIf.h, SchM_NvM.h, MemMap.h, Crc.h and optionally Det.h

Only NvM.h shall be included by the upper layer.

## 5.2 Memory abstraction modules

The memory abstraction modules abstract the NvM module from the subordinated drivers which are hardware dependent. The memory abstraction modules provide a runtime translation of each block access initiated by the NvM module to select the corresponding driver functions which are unique for all configured EEPROM or FLASH storage devices. The memory abstraction module is chosen via the NVRAM block device ID which is configured for each NVRAM block.

## 5.3 CRC module

The NvM module uses CRC generation routines (16/32 bit) to check and to generate CRC for NVRAM blocks as a configurable option. The CRC routines have to be provided externally [ref. to ch. 8.6.2].

## 5.4 Capability of the underlying drivers

A set of underlying driver functions has to be provided for every configured NVRAM device as, for example, internal or external EEPROM or FLASH devices. The unique driver functions inside each set of driver functions are selected during runtime via a memory hardware abstraction module (see chapter 5.2). A set of driver functions has to include all the needed functions to write to, to read from or to maintain (e.g. erase) a configured NVRAM device.

# 6 Requirements traceability

Document: General requirements on Basic Software Modules

| ***Requirement*** | ***Satisfied by*** |
|---|---|
| [BSW00344] Reference to link-time configuration | Not applicable<br>(currently the NVM is only pre-compile-time configurable) |
| [BSW00404] Reference to post build time configuration | Not applicable<br>(currently the NVM is only pre-compile-time configurable) |
| [BSW00405] Reference to multiple configuration sets | Not applicable<br>(currently the NVM is only pre-compile-time configurable) |
| [BSW00345] Pre-compile-time configuration | NVM095 |
| [BSW159] Tool-based configuration | NVM095 |
| [BSW167] Static configuration checking | NVM028, NVM061 |
| [BSW170] Data for reconfiguration of AUTOSAR SW-components | Not applicable<br>(NVM is no AUTOSAR SW-C) |
| [BSW00380] Separate C-File for configuration parameters | Not applicable<br>(currently the NVM is only pre-compile-time configurable) |
| [BSW00419] Separate C-Files for pre-compile time configuration parameters | NVM321 |
| [BSW00381] Separate configuration header file for pre-compile time parameters | NVM028 |
| [BSW00412] Separate H-File for configuration parameters | Not applicable<br>(currently the NVM is only pre-compile-time configurable) |
| [BSW00383] List dependencies of configuration files | NVM095, NVM319, NVM320 |
| [BSW00384] List dependencies to other module | NVM319, NVM320 |
| [BSW00385] List possible error notifications | NVM023, NVM027 |
| [BSW00386] Configuration for detecting an error | NVM023, NVM027, NVM025 |
| [BSW00387] Specify the configuration class of callback functions | NVM330, NVM331 |
| [BSW00388] Introduce containers | NVM028, NVM061, NVM095 |
| [BSW00389] Containers shall have names | NVM028, NVM061, NVM095 |
| [BSW00390] Parameter content shall be unique within the module | NVM028, NVM061, NVM095 |
| [BSW00391] Parameter shall have unique names | NVM028, NVM061, NVM095 |
| [BSW00392] Parameters shall have a type | NVM028, NVM061, NVM095 |
| [BSW00393] Parameters shall have a range | NVM028, NVM061, NVM095 |
| [BSW00394] Specify the scope of the parameters | NVM028, NVM061, NVM095 |
| [BSW00395] List the required parameters (per parameter] | NVM028, NVM061, NVM095 |
| [BSW00396] Configuration classes | NVM028, NVM061, NVM095 |
| [BSW00397] Pre-compile-time parameters | NVM028, NVM061, NVM095 |
| [BSW00398] Link-time-parameters | Not applicable<br>(currently the NVM is only pre-compile-time configurable) |
| [BSW00399] Loadable post-build time parameters | Not applicable<br>(currently the NVM is only pre-compile-time configurable) |

| | |
|---|---|
| [BSW00400] Selectable post-build time parameters | Not applicable<br>(currently the NVM is only pre-compile-time configurable) |
| [BSW00402] Published information | NVM022 |
| [BSW101] Initialization interface | NVM399, NVM400 |
| [BSW00416] Sequence of Initialization | Not applicable<br>(the NvM module isn't responsible for any BSW module initialization) |
| [BSW00406] Check module initialization | NVM399, NVM400, NVM027, NVM023 |
| [BSW003] Version identification | NVM022 |
| [BSW004] Version check | NVM089 |
| [BSW00337] Classification of errors | NVM023, NVM024 |
| [BSW00338] Detection and reporting of development errors | NVM025 |
| [BSW168] Diagnostic Interface | Not applicable<br>(no use case for the NvM module) |
| [BSW00407] Function to read out published parameters | NVM285, NVM286 |
| [BSW00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces | Not applicable |
| [BSW00424] BSW main processing function task allocation [approved] | NVM322 |
| [BSW00425] Trigger conditions for schedulable objects | NVM464 |
| [BSW00426] Exclusive areas in BSW modules | Not applicable |
| [BSW00427] ISR description for BSW modules | Not applicable<br>(NVM doesn't use ISRs) |
| [BSW00428] Execution order dependencies of main processing functions | NVM324 |
| [BSW00429] Restricted BSW OS functionality access | NVM332 |
| [BSW00431] The BSW Scheduler module implements task bodies | Not applicable<br>(Requirement on implementation, not on specification) |
| [BSW00432] Modules should have separate main processing functions for read/receive and write/transmit data path | Not applicable<br>(read and write jobs are handled in a serialized way, not in parallel) |
| [BSW00433] Calling of main processing functions | NVM324 |
| [BSW00434] The Schedule Module shall provide an API for exclusive areas | Not applicable<br>(Requirement for schedule module) |
| [BSW00375] Notification of wake-up reason | Not applicable<br>(no use case) |
| [BSW00339] Reporting of production relevant errors status | NVM026 |
| [BSW00369] Do not return development error codes via API | NVM025 |
| [BSW00421] Reporting of production relevant error events | NVM319 |
| [BSW00422] Debouncing of production relevant error status | Not applicable<br>(NVM uses Dem_ReportError API for error reporting) |
| [BSW00420] Production relevant error event rate detection | Not applicable<br>(Requirement belongs to Dem) |
| [BSW00417] Reporting of Error Events by Non-Basic Software | Not applicable<br>(NVM is a BSW module) |

| | |
|---|---|
| [BSW00409] Header files for production code error IDs | NVM186 |
| [BSW00336] Shutdown interface | Not applicable (no use case) |
| [BSW171] Configurability of optional functionality | NVM028 |
| [BSW00323] API parameter checking | NVM027 |
| [BSW00373] Main processing function naming convention | NVM464 |
| [BSW161] Microcontroller abstraction | Not applicable (Requirement on AUTOSAR architecture, not a single module) |
| [BSW162] ECU layout abstraction | Not applicable (Requirement on AUTOSAR architecture, not a single module) |
| [BSW00324] Do not use HIS I/O Library | Not applicable (architecture decision) |
| [BSW005] No hard coded horizontal interfaces within MCAL | Not applicable (Requirement on AUTOSAR architecture, not a single module) |
| [BSW00415] User dependent include files | Not applicable (NVM doesn't provide restricted access for several modules) |
| [BSW164] Implementation of interrupt service routines | Not applicable (this module doesn't implement any ISRs) |
| [BSW00325] Runtime of interrupt service routines | Not applicable (this module doesn't implement any ISRs) |
| [BSW00326] Transition from ISRs to OS tasks | Not applicable (this module doesn't implement any ISRs) |
| [BSW00342] Usage of source code and object code | Not applicable (Requirement on AUTOSAR architecture, not a single module) |
| [BSW00343] Specification and configuration of time | Not applicable (no configurable timings) |
| [BSW160] Human-readable configuration data | Not applicable (Requirement on documentation, not on specification) |
| [BSW007] HIS MSIRA C | Not applicable (Requirement on implementation, not on specification) |
| [BSW00300] Module naming convention | Chapter 5.1 |
| [BSW00413] Accessing instances of BSW modules | Conflict: This requirement will have impact on almost all BSW modules, therefore it can not be implemented within the Release 2.0 timeframe. |
| [BSW00347] Naming separation of different instances of BSW drivers | Not applicable (Requirement on implementation, not on specification) |
| [BSW00305] Self-defined data types naming convention | Chapter 8.2 |
| [BSW00307] Global variables naming convention | Not applicable (Requirement on implementation, not on specification) |
| [BSW00310] API naming convention | Chapters 8.3, 0, 0 |
| [BSW00327] Error values naming convention | NVM023, NVM027 |
| [BSW00335] Status values naming convention | Not applicable (no status values available) |
| [BSW00350] Development error detection | NVM025, NVM188 |

| | |
|---|---|
| keyword | |
| [BSW00408] Configuration parameter naming convention | Chapter 10 |
| [BSW00410] Compiler switches shall have defined values | Chapter 10 |
| [BSW00411] Get version info keyword | NVM286, Chapter 10 |
| [BSW00346] Basic set of module files | Chapter 5.1 |
| [BSW158] Separation of configuration from implementation | Chapter 5.1 |
| [BSW00314] Separation of interrupt frames and service routines | Not applicable<br>(this module doesn't implement any ISRs) |
| [BSW00370] Separation of callback interface from API | Chapter 5.1 |
| [BSW00348] Standard type header | Not applicable<br>(this module simply includes the standard type header via the module header file) |
| [BSW00353] Platform specific type header | Not applicable<br>(this module simply includes the standard type header via the module header file) |
| [BSW00361] Compiler specific language extension header | Not applicable<br>(this module simply includes the standard type header via the module header file) |
| [BSW00301] Limit imported information | Chapter 5.1 |
| [BSW00302] Limit exported information | Not applicable<br>(Requirement on the implementation, not on the specification) |
| [BSW00328] Avoid duplication of code | Not applicable<br>(Requirement on the implementation, not on the specification) |
| [BSW00312] Shared code shall be reentrant | Not applicable<br>(Requirement on the implementation, not on the specification) |
| [BSW006] Platform independency | Not applicable<br>(Requirement on the implementation, not on the specification) |
| [BSW00357] Standard API return type | Chapter 8.1, 8.2 |
| [BSW00377] Module specific API return types | Chapter 8.2 |
| [BSW00304] AUTOSAR integer data types | Not applicable<br>(Requirement on implementation, not for specification) |
| [BSW00355] Do not redefine AUTOSAR integer data types | Not applicable<br>(Requirement on implementation, not for specification) |
| [BSW00378] AUTOSAR boolean type | Not applicable<br>(Requirement on implementation, not for specification) |
| [BSW00306] Avoid direct use of compiler and platform specific keywords | Not applicable<br>(Requirement on implementation, not for specification) |
| [BSW00308] Definition of global data | Not applicable<br>(Requirement on implementation, not for specification) |
| [BSW00309] Global data with read-only constraint | Not applicable<br>(Requirement on implementation, not for specification) |
| [BSW00371] Do not pass function pointers via API | Not applicable |

| | |
|---|---|
| | (no function pointers in this specification) |
| [BSW00358] Return type of init() functions | Chapter 8.3.1.1 |
| [BSW00414] Parameter of init function | Chapter 8.3.1.1 |
| [BSW00376] Return type and parameters of main processing functions | NVM106 |
| [BSW00359] Return type of callback functions | Chapter 8.6.3.1, 0 |
| [BSW00360] Parameters of callback functions | Chapter 8.6.3.1, 0 |
| [BSW00329] Avoidance of generic interfaces | Chapter 8.3 |
| [BSW00330] Usage of macros / inline functions instead of functions | Not applicable<br>Requirement on implementation, not for specification) |
| [BSW00331] Separation of error and status values | NVM023, NVM027 |
| [BSW009] Module User Documentation | Not applicable<br>(Requirement on documentation, not on specification) |
| [BSW00401] Documentation of multiple instances of configuration parameters | Chapter 10 |
| [BSW172] Compatibility and documentation of scheduling strategy | NVM323, NVM324 |
| [BSW010] Memory resource documentation | Not applicable<br>(Requirement on documentation, not on specification) |
| [BSW00333] Documentation of callback function context | Chapter 8.6.3.1, 0 |
| [BSW00374] Module vendor identification | NVM022 |
| [BSW00379] Module identification | NVM022 |
| [BSW003] Version identification | NVM022 |
| [BSW00318] Format of module version numbers | NVM022 |
| [BSW00321] Enumeration of module version numbers | Not applicable<br>(Requirement on implementation, not for specification) |
| [BSW00341] Microcontroller compatibility documentation | Not applicable<br>(Requirement on documentation, not on specification) |
| [BSW00334] Provision of XML file | Not applicable<br>(Requirement on documentation, not on specification) |
| [BSW00435] Header File Structure for the Basic Software Scheduler | NVM077 |
| [BSW00436] Module Header File Structure for the Basic Software Memory Mapping | NVM077 |

## Document: Requirements on Memory Services

| ***Requirement*** | ***Satisfied by*** |
|---|---|
| [BSW041] Declaration and allocation of application memory | NVM030 |
| [BSW08534] Classes of RAM data blocks | NVM370, NVM371, NVM372, NVM373 |
| [BSW08528] NVRAM block management type - native | NVM000 |
| [BSW08529] NVRAM block management type - redundant | NVM001, NVM047 |
| [BSW08531] NVRAM block management type – dataset | NVM006 |
| [BSW08543] Static configuration of block priority | NVM032 |

| | |
|---|---|
| [BSW08009] Default write protection of blocks | NVM033, NVM054 |
| [BSW135] NVRAM configuration ID | NVM034, NVM073 |
| [BSW08549] Automatic initialization of RAM data after Software update | NVM116 |
| [BSW125] Job notification | NVM383, NVM384, |
| [BSW08000] Configurable access to multiple (different) devices | NVM035 |
| [BSW08001] Configuration of consistency check of data | NVM036, NVM040 |
| [BSW08538] Static configuration of NVRAM blocks being loaded during start-up | NVM117, NVM245, NVM118 |
| [BSW08546] Protection of RAM data blocks against data loss [approved] | NVM119 |
| [BSW08533] Load data blocks from NVRAM to RAM | NVM008 |
| [BSW176] Only access non-volatile memory via NVRAM manager | NVM037<br>(note: this has to be handled on RTE level) |
| [BSW027] Accessing of non volatile data | NVM038 |
| [BSW08014] RAM block allocation | NVM088 |
| [BSW013] Handling of concurrent accesses to NVRAM | NVM378, NVM379 |
| [BSW016] Block-wise reading of data | NVM010 NVM029 |
| [BSW017] Block-wise writing of data | NVM410, NVM411 |
| [BSW08541] Guaranteed processing of accepted write requests | NVM380, NVM381, NVM152 |
| [BSW018] Block-wise restoring of default data | NVM012 |
| [BSW08548] Automatic initialization without ROM Block | NVM116 |
| [BSW08547] Distinction between invalidated and inconsistent data | NVM405, NVM406, NVM294, NVM203, |
| [BSW08550] Marking blocks modified/unmodified | NVM405, NVM406, NVM432, NVM433, NVM434, NVM344, NVM345 |
| [BSW08545] Validation of permanent RAM data blocks | NVM405, NVM406, NVM121, NVM344, NVM345 |
| [BSW08011] Invalidation of NVRAM data blocks | NVM421, NVM422, NVM423, NVM424 |
| [BSW08544] Block-wise erasing of NVRAM data | NVM415, NVM416, NVM417, NVM418 |
| [BSW08007] Selection of datasets | NVM014, NVM021 |
| [BSW08542] Job order prioritization | NVM032 |
| [BSW020] Readout of current status of NVRAM manager operations | NVM015 |
| [BSW127] Write protect/unprotect function | NVM016, NVM054 |
| [BSW030] Consistency/integrity check of data | NVM040, NVM036, NVM165 |
| [BSW034] Quasi-parallel write access | NVM378, NVM379 |
| [BSW08535] Save data blocks from RAM to NV memory | NVM018 |
| [BSW08540] Aborting the shut down process | NVM019 |
| [BSW038] Treatable errors shall not affect other software components | NVM047, NVM001 |
| [BSW129] Automatic data repair | NVM047, NVM001 |
| [BSW08010] Loading of ROM default data | NVM012, NVM020, NVM387, NVM388 |
| [BSW011] (Memory) hardware independence | NVM051 |
| [BSW130] Provide information about used memory resources | NVM052 |
| [BSW08015] NV Block security of ECU reprogramming | NVM072, NVM276 |

# 7 Functional specification

## 7.1 Basic architecture guidelines

### 7.1.1 Layer structure

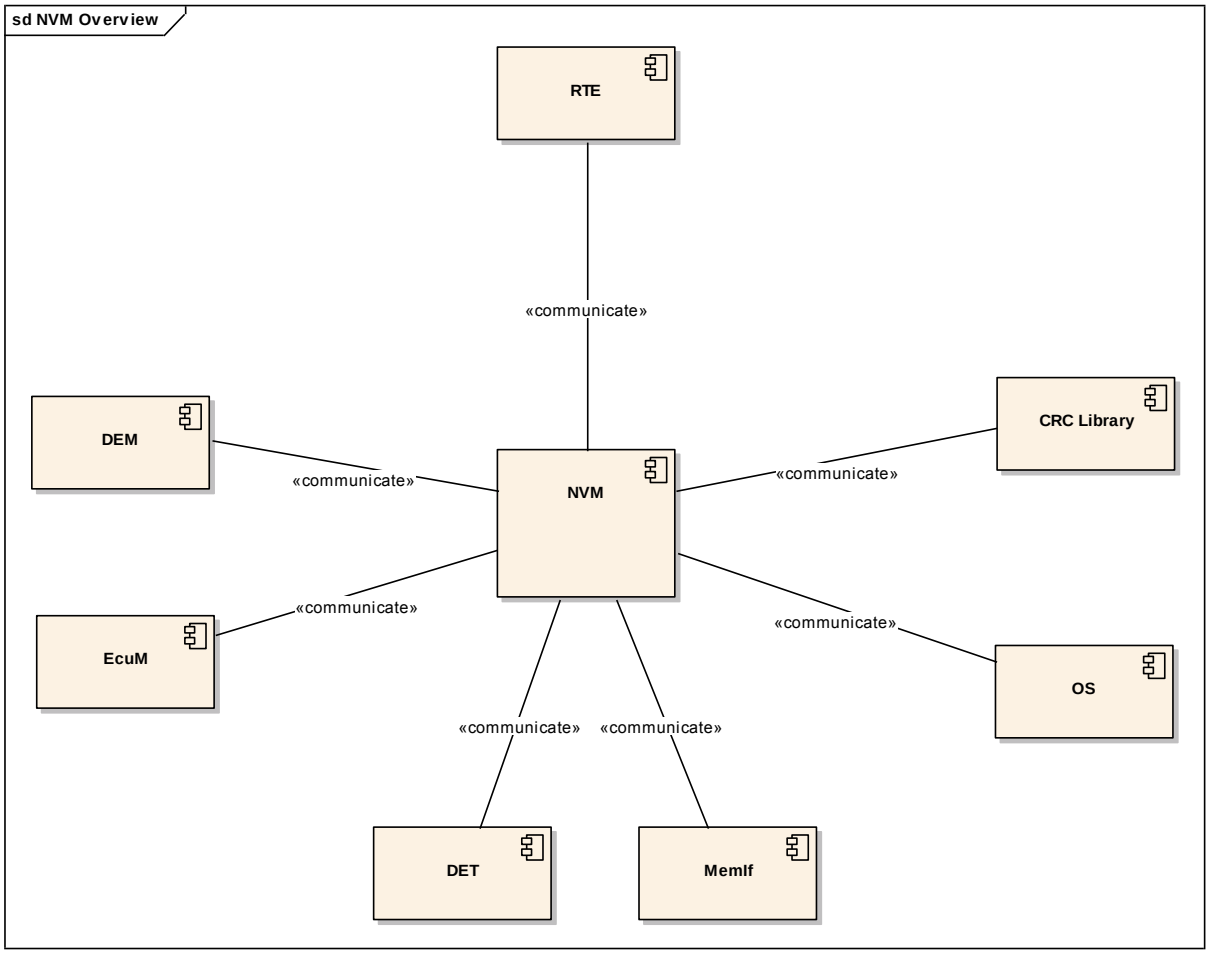


**Figure 2: NVRAM Manager interactions overview**

### 7.1.2 Addressing scheme for the memory hardware abstraction

**NVM051**: The Memory Abstraction Interface, the underlying Flash EEPROM Emulation and EEPROM Abstraction Layer provide the NvM module with a virtual linear 32bit address space which is composed of a 16bit logical block number and a 16bit block address offset.

Hint: According to NVM051, the NvM module can allow for a (theoretical) number of 65536 logical blocks, each logical block can have a (theoretical)[1] size of up to 64 Kbytes.

**NVM122**: The NvM module shall further subdivide the 16bit logical block number into the following parts:

- block identifier with the size of (16 - `NvmDatasetSelectionBits`) bits
- `NvmDatasetSelectionBits` bit data index, allowing for up to 256 datasets per NVRAM block

**NVM343**: Handling/addressing of redundant NVRAM blocks shall be done towards the memory hardware abstraction in the same way like for dataset NVRAM blocks, i.e. the redundant NV blocks shall be managed by usage of the configuration parameter `NvmDatasetSelectionBits`.

**NVM123**: The NvM module shall store the block identifier in the most significant bits of the 16bit logical block number.

**NVM442:** The configuration tool shall configure the block identifiers.

**NVM443:** The NvM module shall not modify the configured block identifiers.

#### 7.1.2.1 Examples of addressing scheme for the memory hardware abstraction

To clarify the previously described addressing scheme which is used for NVRAM manager ↔ memory hardware abstraction interaction, the following examples shall help to understand the correlations between the configuration parameters `NvmNvBlockBaseNumber`, `NvmDatasetSelectionBits` on NVRAM manager side and `EA_BLOCK_NUMBER` / `FEE_BLOCK_NUMBER` on memory hardware abstraction side [NVM061].

For the given examples A and B a simple formula is used:
`FEE/EA_BLOCK_NUMBER = NvmNvBlockBaseNumber + NvmDatasetSelectionBits.`

Example A:
The configuration parameter `NvmDatasetSelectionBits` is configured to be 2. This leads to the result that 14 bits are available as range for the configuration parameter `NvmNvBlockBaseNumber`.
⇒ Range of `NvmNvBlockBaseNumber`: 0x1..0x3FFE
⇒ Range of `NvmDatasetSelectionBits`: 0x0..0x3
⇒ Range of `FEE_BLOCK_NUMBER/EA_BLOCK_NUMBER`: 0x4..0xFFFB

Example B:

[1] "Theoretical" meaning here that we don't expect anyone to use the NVRAM manager and underlying layers in exactly this way.

The configuration parameter `NvmDatasetSelectionBits` is configured to be 4. This leads to the result that 12 bits are available as range for the configuration parameter `NvmNvBlockBaseNumber`.

⇒ Range of `NvmNvBlockBaseNumber`: 0x1..0xFFE

⇒ Range of `NvmDatasetSelectionBits`: 0x0..0xF

⇒ Range of `FEE/EA Block Number`: 0x10..0xFFEF

### 7.1.3 Basic storage objects

#### 7.1.3.1 NV block

**NVM125:** The NV block is a basic storage object and represents a memory area consisting of NV user data and (optionally) a CRC value.

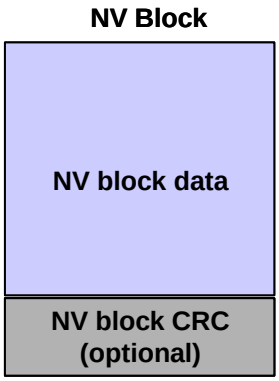


**Figure 3: NV Block layout**

Note: This figure does not show the physical memory layout of an NV block. Only the logical clustering is shown.

#### 7.1.3.2 RAM block

**NVM126:** The RAM block is a basic storage object and represents an area in RAM consisting of user data and (optionally) a CRC value.

**NVM127:** Restrictions on CRC usage on RAM blocks. CRC is only available if the corresponding NV block(s) also have a CRC. CRC has to be of the same type as that of the corresponding NV block(s). [NVM061]

**NVM129:** The user data area of a RAM block can reside in a different RAM address location (global data section) than the state of the RAM block**.**
**NVM130**: The data area of a RAM block shall be accessible from NVRAM Manager and from the application side (data passing from/to the corresponding NV block).

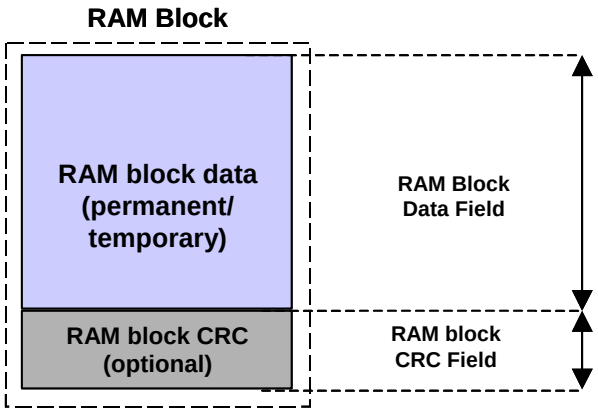


**Figure 4:** RAM Block layout

Note: This figure does not show the physical memory layout of a RAM block. Only the logical clustering is shown.
As the NvM module doesn't support alignment, this could be managed by configuration, i.e. the block length could be enlarged by adding padding to meet alignment requirements.

**NVM373:** The RAM block data shall contain the permanently or temporarily assigned user data.

**NVM371:** The RAM block data is assigned to exactly one SW-Component or BSW module.

**NVM370:** In case of permanently assigned user data, the address of the RAM block data is known during configuration time.

**NVM372:** In case of temporarily assigned user data, the address of the RAM block data is not known during configuration time and will be passed to the NvM module during runtime. The RAM block data can be used e.g. as a shared buffer for several SW-Components or BSW modules.

**NVM088**: It shall be possible to allocate each RAM block without address constraints in the global RAM area. The whole number of configured RAM blocks needs not be located in a continuous address space.

#### 7.1.3.3 ROM block

**NVM020**: The ROM block is a basic storage object, resides in the ROM (FLASH) and is used to provide default data in case of an empty or damaged NV block**.**

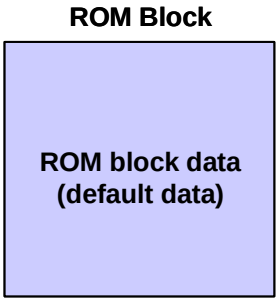


**Figure 5: ROM block layout**

#### 7.1.3.4 Administrative block

**NVM134**: The Administrative block shall be located in RAM and shall contain a block index which is used in association with Dataset NV blocks. Additionally, attribute/error/status information of the corresponding NVRAM block shall be contained.

**NVM128**: The NvM module shall use state information of the permanent RAM block or of the RAM mirror in the NvM module in case of explicit syncronization (invalid/valid) to determine the validity of the permanent RAM block user data.

**NVM132**: The RAM block state „invalid“ indicates that the data area of the respective RAM block is invalid. The RAM block state „valid“ indicates that the data area of the respective RAM block is valid.

**NVM133**: The value of “invalid” shall be represented by all other values except “valid”.

**NVM135**: The Administrative block shall be invisible for the application and is used exclusively by the NvM module for security and administrative purposes of the RAM block and the NVRAM block itself.

**NVM054**: The NvM module shall use an attribute field to manage the NV block write protection in order to protect/unprotect a NV block data field.

**NVM136**: The NvM module shall use an error/status field to manage the error/status value of the last request [NVM083].

### 7.1.4 Block management types

#### 7.1.4.1 Block management types overview

**NVM137**: The following types of NVRAM storage shall be supported by the NvM module implementation and consist of the following basic storage objects:

| *Management Type* | *NV Blocks* | *RAM Blocks* | *ROM Blocks* | *Administrative Blocks* |
|---|---|---|---|---|
| NVM_BLOCK_NATIVE | 1 | 1 | 0..1 | 1 |
| NVM_BLOCK_REDUNDANT | 2 | 1 | 0..1 | 1 |
| NVM_BLOCK_DATASET | 1..(m<256)* | 1 | 0..n | 1 |

* The number of possible datasets depends on the configuration parameter `NvmDatasetSelectionBits`.

#### 7.1.4.2 NVRAM block structure

**NVM138**: The NVRAM block shall consist of the mandatory basic storage objects NV block, RAM block and Administrative block.

**NVM139**: The basic storage object ROM block is optional.

**NVM140**: The composition of any NVRAM block is fixed during configuration by the corresponding NVRAM block descriptor.

**NVM141**: All address offsets are given relatively to the start addresses of RAM or ROM in the NVRAM block descriptor. The start address is assumed to be zero. A device specific base address or offset will be added by the respective device driver if needed.

For details of the NVRAM block descriptor see chapter 7.1.4.3.

#### 7.1.4.3 NVRAM block descriptor table

**NVM069:** A single NVRAM block to deal with will be selected via the NvM module API by providing a subsequently named Block ID**.**

**NVM143:** All structures related to the NVRAM block descriptor table and their addresses in ROM (FLASH) have to be generated during configuration of the NvM module.

#### 7.1.4.4 Native NVRAM block

The Native NVRAM block is the simplest block management type. It allows storage to/retrieval from NV memory with a minimum of overhead.

**NVM000:** The Native NVRAM block consists of a single NV block, RAM block and Administrative block**.**

### 7.1.4.5 Redundant NVRAM block

In addition to the Native NVRAM block, the Redundant NVRAM block provides enhanced fault tolerance, reliability and availability. It increases resistance against data corruption.

**NVM001:** The Redundant NVRAM block consists of two NV blocks, a RAM block and an Administrative block**.**

The following figure reflects the internal structure of the NV memory:

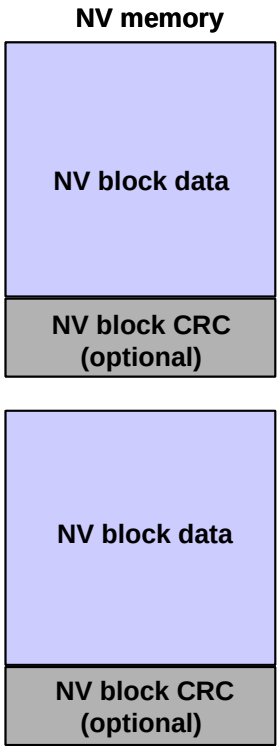


**Figure 6: Redundant NVRAM Block layout**

Note: This figure does not show the physical NV memory layout of a redundant NVRAM block. Only the logical clustering is shown.

### 7.1.4.6 Dataset NVRAM block

The Dataset NVRAM block is an array of equally sized data blocks (NV/ROM). The application can at one time access exactly one of these elements.

**NVM006**: The Dataset NVRAM block consists of multiple NV user data and (optionally) CRC areas, a RAM block and an Administrative block.

**NVM144**: The index position of the dataset is noticed via a separated field in the corresponding Administrative block.

**NVM374**: The NvM module shall be able to read all assigned NV blocks.

**NVM375:** The NvM module shall only be able to write to all assigned NV blocks if (and only if) write protection is disabled.

**NVM146**: If the basic storage object ROM block is selected as optional part, the index range which normally selects a dataset is extended to the ROM to make it possible to select a ROM block instead of a NV block**.** The index covers all NV/ROM blocks which may build up the NVRAM Dataset block.

**NVM376:** The NvM module shall be able to only read optional ROM blocks (default datasets).

**NVM377:** The NvM module shall treat a write to a ROM block like a write to a protected NV block**.**
**NVM444:** The total number of configured datasets (NV+ROM blocks) must be in the range of 1..255.

**NVM445:** In case of optional ROM blocks, data areas with an index from 0 up to NvmNvBlockNum-1 represent the NV blocks with their CRC in the NV memory**.** Data areas with an index from NvmNvBlockNum up to NvmNvBlockNum+NvmRomBlockNum represent the ROM blocks.

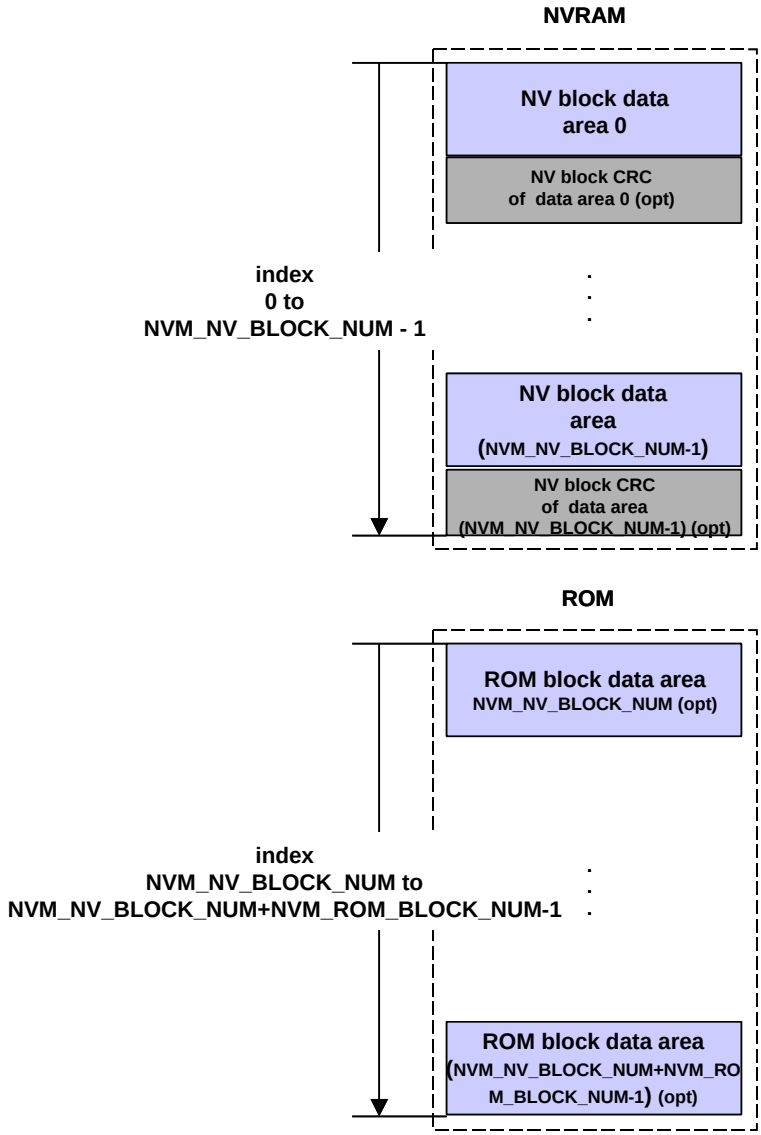


**Figure 7: Dataset NVRAM block layout**

Note: This figure does not show the physical NV memory layout of a Dataset NVRAM block. Only the logical clustering is shown.

#### 7.1.4.7 NVRAM Manager API configuration classes

**NVM149:** To have the possibility to adapt the NvM module to limited hardware resources, three different API configuration classes shall be defined:

- **API configuration class 3:**
  All specified API calls are available. A maximum of functionality is supported.

- **API configuration class 2:**
  An intermediate set of API calls is available.

- **API configuration class 1:**
  Especially for matching systems with very limited hardware resources this API configuration class offers only a minimum set of API calls which are required in any case.

| ***API configuration class 3*** | ***API configuration class 2*** | ***API configuration class 1*** |
|---|---|---|
| Type 1:<br><br>NvM_SetDataIndex (...)<br>NvM_GetDataIndex (...)<br>NvM_SetBlockProtection (...)<br>NvM_GetErrorStatus(...)<br>NvM_SetRamBlockStatus(...) | Type 1:<br><br>NvM_SetDataIndex (...)<br>NvM_GetDataIndex (...)<br>NvM_GetErrorStatus(...)<br>NvM_SetRamBlockStatus(...) | Type 1:<br><br>NvM_GetErrorStatus(...)<br>NvM_SetRamBlockStatus(...) |
| Type 2:<br><br>NvM_ReadBlock(...)<br>NvM_WriteBlock(...)<br>NvM_RestoreBlockDefaults(...)<br>NvM_EraseNvBlock(...)<br>NvM_InvalidateNvBlock(...) | Type 2:<br><br>NvM_ReadBlock(...)<br>NvM_WriteBlock(...)<br>NvM_RestoreBlockDefaults(...) | Type 2:<br><br>-- |
| Type 3:<br><br>NvM_ReadAll(...)<br>NvM_WriteAll(...)<br>NvM_CancelWriteAll(...) | Type 3:<br><br>NvM_ReadAll(...)<br>NvM_WriteAll(...)<br>NvM_CancelWriteAll(...) | Type 3:<br><br>NvM_ReadAll(...)<br>NvM_WriteAll(...)<br>NvM_CancelWriteAll(...) |
| Type 4:<br><br>NvM_Init(...) | Type 4:<br><br>NvM_Init(...) | Type 4:<br><br>NvM_Init(...) |

Note: For API configuration class 1 no queues are needed, no immediate data can be written. Furthermore the API call `NvM_SetRamBlockStatus` is only available if configured by `NvmSetRamBlockStatusApi`.

**NVM365:** Within API configuration class 1, the block management type NVM_BLOCK_DATASET is not supported.

For information regarding the definition of Type 1…4 ref. to chapter 8.7.

**NVM150:** The NvM module shall only contain that code that is needed to handle the configured block types.

### 7.1.5 Scan order / priority scheme

**NVM032:** The NvM module shall support a priority based job processing. By configuration parameter `NvmJobPrioritization` [NVM028] priority based job processing shall be enabled/disabled.

**NVM378:** In case of priority based job processing order, the NvM module shall use two queues, one for immediate write jobs (crash data) another for all other jobs (including immediate read/erase jobs).

**NVM379:** If priority based job processing is disabled via configuration, the NvM module shall not support immediate write jobs. In this case, the NvM module processes all jobs in FCFS order.

**NVM380:** The job queue length for multi block requests originating from the `NvM_ReadAll` and `NvM_WriteAll` shall be one (only one job is queued).

**NVM381:** The NvM module shall not interrupt jobs originating from the `NvM_ReadAll` and `NvM_WriteAll` request by other requests. The NvM module shall rather queue read / write jobs that are requested during an ongoing `NvM_ReadAll` / `NvM_WriteAll` request and executed them subsequently.

Note: The `NvM_WriteAll` request can be aborted by calling `NvM_CancelWriteAll`. In this case, the current block is processed completely but no further blocks are written [NVM238].

Hint: It shall be allowed to dequeue requests, if they became obsolete by completion of the regarding NVRAM block.

**NVM152**: The only exception to the rule given in [NVM380, NVM381] is a write job with immediate priority which shall preempt the running read / write job. The preempted job shall subsequently be resumed / restarted by the NvM module.

## 7.2 General behavior

### 7.2.1 Functional requirements

**NVM383:** For each asynchronous request, a notification of the caller after completion of the job shall be a configurable option.

**NVM384:** The NvM module shall provide a callback interface [NVM113].

**NVM037:** The NvM module's environment shall access the non-volatile memory via the NvM module only. It shall not be allowed for any module (except for the NvM module) to access the non-volatile memory directly.

**NVM038:** The NvM module only provides an implicit way of accessing blocks in the NVRAM and in the shared memory (RAM). This means, the NvM module copies one or more blocks from NVRAM to the RAM and the other way round.

The application accesses the RAM data directly, with respect to given restrictions (e.g. synchronization).

**NVM385:** The NvM module shall queue all asynchronous “single block” read/write/control requests if the block with its specific ID is not already queued or currently in progress (multitasking restrictions).

**NVM386:** The NvM module shall accept multiple asynchronous “single block” requests as long as no queue overflow occurs.

**NVM155**: The highest priority request shall be fetched from the queues by the NvM module and processed in a serialized order.

**NVM040**: The NvM module shall implement implicit mechanisms for consistency / integrity checks of data saved in NV memory [NVM165].

**NVM156**: Depending on implementation, callback routines provided and/or invoked by the NvM module may be called in interrupt context. The NvM module providing those routines has therefore to make sure that their runtime is reasonably short.

**NVM042**: The NvM module shall be able to detect corrupted / invalid data during `NvM_ReadAll` by performing a checksum calculation and/or testing the RAM block validity, which is managed in the administrative block.[NVM036] [NVM008]

**NVM085**: If there is no default ROM data available at configuration time or no callback defined by `NvmInitBlockCallback` then the application shall be responsible for providing the default initialization data.

In this case, the application has to use `NvM_GetErrorStatus()` to be able to distinguish [NVM061] between first initialization and corrupted data. [NVM083]

**NVM387:** During processing of `NvM_ReadAll`, the NvM module shall be able to detect corrupted / invalid RAM data by performing a checksum calculation and/or testing the validity of a data within the administrative block.

**NVM388:** During startup phase and normal operation of `NvM_ReadAll` and if the NvM module has detected an unrecoverable error within the NV block, the NvM module shall copy default data (if configured) to the corresponding RAM block.

**NVM332:** The NvM module shall use the BSW scheduler only, i.e. instead of directly making use of OS objects and/or related OS services.

### 7.2.2 Design notes

#### 7.2.2.1 NVRAM manager startup

`NvM_Init` shall be invoked by the ECU state manager exclusively.

**NVM091:** Due to strong constraints concerning the ECU startup time, the `NvM_Init` request shall not contain the initialization of the configured NVRAM blocks.

**NVM157:** The `NvM_Init` request shall not be responsible to trigger the initialization of underlying drivers and memory hardware abstraction. This shall also be handled by the ECU state manager.

**NVM158:** The initialization of the RAM data blocks shall be done by another request, namely `NvM_ReadAll` [NVM008].

`NvM_ReadAll` shall be called exclusively by the ECU state manager.

Software components which use the NvM module shall be responsible for checking global error/status information resulting from the NvM module startup. The ECU state manager shall use polling by using `NvM_GetErrorStatus` [NVM015] (reserved block ID 0) or callback notification (configurable option `NvmMultiBlockCallback` [NVM028]) to derive global error/status information resulting from startup. If polling is used, the end of the NVRAM startup procedure shall be detected by the global error/status `NVM_REQ_OK` or `NVM_REQ_NOT_OK` (during startup `NVM_REQ_PENDING`) [NVM083]. If callbacks are chosen for notification, software components shall be notified automatically if an assigned NVRAM block has been processed.[NVM281].

Note 1: If callbacks are configured for each NVRAM block which is processed within `NvM_ReadAll`, they can be used by the RTE to start e.g. SW-Cs at an early point of time.

Note 2: To ensure that the DEM is fully operational at an early point of time, i.e. its NV data is restored to RAM, DEM related NVRAM blocks should be configured to have a low ID to be processed first within `NvM_ReadAll`.

**NVM160:** The NvM module shall not store the currently used Dataset index automatically in a persistent way.

Software components shall check the specific error/status of all blocks they are responsible for by using `NvM_GetErrorStatus` [NVM015] with specific block IDs to determine the validity of the corresponding RAM blocks.

For all blocks of the block management type “NVRAM Dataset” [NVM006] the software component shall be responsible to set the proper index position by `NvM_SetDataIndex` [NVM014]. E.g. the current index position can be stored/maintained by the software component in a unique NVRAM block. To get the current index position of a “Dataset Block”, the software component shall use the `NvM_GetDataIndex` [NVM021] API call.

#### 7.2.2.2 NVRAM manager shutdown

**NVM092:** The basic shutdown procedure shall be done by the request `NvM_WriteAll` [NVM018].

Hint: `NvM_WriteAll` shall be invoked by the ECU state manager.

#### 7.2.2.3 (Quasi) parallel write access to the NvM module

**NVM162:** The NvM module shall receive the requests via an asynchronous interface using a queuing mechanism. The NvM module shall process all requests serially depending on their priority.

#### 7.2.2.4 Avoid infinite loops

**NVM163:** The DEM is responsible for handling the loop detection.

#### 7.2.2.5 NVRAM block consistency check

**NVM164:** The NvM module shall provide implicit techniques to check the data consistency of NVRAM blocks [NVM036], [NVM040]. The data consistency check of a NVRAM block shall be done by CRC recalculations of its corresponding NV block(s).

**NVM165:** The implicit way of a data consistency check shall be provided by configurable options of the internal functions. The implicit consistency check shall be configurable for each NVRAM block and depends on the configurable parameters `NvmBlockUseCrc` and `NvmCalcRamBlockCrc` [NVM061].

**NVM744:** Depending on the configurable parameters `NvMBlockUseCrc` and `NvMCalcRamBlockCrc`, NvM module shall allocate memory per block corresponding to configured CRC size.

Hint:
NvM users must not know anything about CRC memory (e.g. size, location) for their data in a RAM block.

#### 7.2.2.6 Error recovery

**NVM047:** The NvM module shall provide techniques for error recovery. The error recovery depends on the NVRAM block management type [NVM001].

**NVM389:** The NvM module shall provide errror recovery on read for every kind of NVRAM block management type by loading of default values.

**NVM390:** The NvM module shall provide error recovery on read for NVRAM blocks of block management type `NVM_BLOCK_REDUNDANT` by loading the redundant NV data with default values.

**NVM168:** The NvM module shall provide error recovery on write by performing write retries regardless of the NVRAM block management type.

**NVM169:** The NvM module shall provide read error recovery on startup for all NVRAM blocks with configured RAM CRC in case of RAM block revalidation failure.

#### 7.2.2.7 Recovery of a RAM block with ROM data

**NVM171:** The NvM module shall provide implicit and explicit recovery techniques to restore ROM data to its corresponding RAM block in case of unrecoverable data inconsistency of a NV block [NVM387, NVM388].

Application hint:
As the NvM module does not provide a mechanism or special status information to inform the caller that a ROM block has been loaded due to recovery of a RAM block, this has to be managed by e.g. SW-Cs itself. A possible solution could be to add additional data marking the block as ROM defaults.

#### 7.2.2.8 Implicit recovery of a RAM block with ROM default data

**NVM172**: The implicit recovery shall be provided during startup (part of `NvM_ReadAll`) and `NvM_ReadBlock` for each NVRAM block with a configured ROM block and a permanent RAM block. If permanent RAM block and NV block are invalid or inconsistent, operations shall be performed as specified in the table below. The data content of the corresponding NV block shall remain unmodified.

| *NVRAM block configured with ROM block* | *per. RAM block state + CRC* | *NV block state* | *read attempt from NV fails* | *Actions done regarding per. RAM* |
|---|---|---|---|---|
| no | -- | -- | -- | no ROM data loaded |
| yes | valid + consistent | -- | -- | no ROM data loaded |
| yes | invalid + (in)consistent | valid | no | no ROM data loaded |
| yes | invalid + (in)consistent | valid | yes | ROM data loaded |
| yes | invalid + (in)consistent | invalid | -- | ROM data loaded |

#### 7.2.2.9 Explicit recovery of a RAM block with ROM default data

**NVM391:** For explicit recovery with ROM block data the NvM module shall provide a function `NvM_RestoreBlockDefaults` [NVM012] to restore ROM data to its corresponding RAM block.

**NVM392:** The function `NvM_RestoreBlockDefaults` shall remain unmodified the data content of the corresponding NV block.

Hint:
The function `NvM_RestoreBlockDefaults` shall be used by the application to restore ROM data to the corresponding RAM block every time it is needed.

#### 7.2.2.10 Detection of an incomplete write operation to a NV block

**NVM174:** The detection of an incomplete write operation to a NV block is out of scope of the NvM module. This is handled and detected by the memory hardware abstraction. The NvM module expects to get information from the memory hardware abstraction if a referenced NV block is invalid or inconsistent and cannot be read when requested.
SW-Cs may use NvM_InvalidateNvBlock to prevent lower layers from delivering old data.

#### 7.2.2.11 Termination of a single block request

**NVM175:** All asynchronous requests provided by the NvM module (except for `NvM_CancelWriteAll`) shall indicate their result in the designated error/status field of the corresponding Administrative block [NVM000].

**NVM176:** The optional configuration parameter `NvmSingleBlockCallback` configures the notification via callback on the termination of an asynchronous block request (except for `NvM_CancelWriteAll`) [NVM061].

#### 7.2.2.12 Termination of a multi block request

**NVM393:** The NvM module shall use a separate variable to store the result of an asynchronous multi block request (`NvM_ReadAll`, `NvM_WriteAll` including `NvM_CancelWriteAll`).

**NVM394:** The function `NvM_GetErrorStatus` [NVM015] shall return the most recent error/status information of an asynchronous multi block request (including `NvM_CancelWriteAll`) [NVM083] in conjunction with a reserved block ID value of 0.

**NVM395:** The result of a multi block request shall represent only a common error/status information.

**NVM396:** The multi block requests provided by the NvM module shall indicate their detailed error/status information in the designated error/status field of each affected Administrative block.

**NVM179:** The optional configuration parameter `NvmMultiBlockCallback` configures the notification via callback on the termination of an asynchronous multi block request [NVM028].

#### 7.2.2.13 General handling of asynchronous requests/ job processing

**NVM180:** Every time when CRC calculation is processed within a request, the NvM module shall calculate the CRC in multiple steps if the referenced NVRAM block length exceeds the number of bytes configured by the parameter `NvmCrcNumOfBytes`.

**NVM351:** For CRC calculation, the NvM module shall use initial values which are published by the CRC module.

**NVM181:** Multiple concurrent single block requests shall be queueable.

**NVM182:** The NvM module shall interrupt asynchronous request/job processing in favor of jobs with immediate priority (crash data).

**NVM184:** If the invocation of an asynchronous function on the NvM module leads to a job queue overflow, the function shall return with `E_NOT_OK`.

**NVM185:** On successful enqueuing a request, the NvM module shall set the request result of the corresponding NVRAM block to `NVM_REQ_PENDING`.

**NVM270:** If the NvM module has successfully processed a job, it shall return `NVM_REQ_OK` as job result.

#### 7.2.2.14 NVRAM block write protection

The NvM module shall offer different kinds of write protection which shall be configurable. Every kind of write protection is only related to the NV part of NVRAM block, i.e. the RAM block data can be modified but not be written to NV memory.

**NVM325:** For enabling and disabling write protection the function `NvM_SetBlockProtection` [NVM016] shall be used which depends on the configuration parameters `NvmBlockWriteProt` and `NvmWriteBlockOnce`.

| ***NvM_SetBlockProtection can modify write protection?*** | ***NvmBlockWriteProt*** | ***NvmWriteBlockOnce*** |
|---|---|---|
| YES | TRUE | FALSE |
| YES | FALSE | FALSE |

| | | |
|---|---|---|
| NO | TRUE | TRUE |
| NO | FALSE | TRUE |

**NVM326:** For all NVRAM blocks configured with `NvmBlockWriteProt == TRUE`, the NvM module shall enable a default write protection. The NvM module's environment can explicitly disable the write protection using the `NvM_SetBlockProtection` function.

**NVM397:** For NVRAM blocks configured with `NvmWriteBlockOnce == TRUE` [NVM072], the NvM module shall only write once to the associated NV memory, i.e in case of a blank NV device.

**NVM398:** For NVRAM blocks configured with `NvmWriteBlockOnce == TRUE`, the NvM module shall not allow disabling the write protection explicitly using the `NvM_SetBlockProtection` function.[NVM276]

#### 7.2.2.15 Validation and modification of RAM block data

This chapter shall give summarized information regarding the internal handling of NVRAM Manager status bits. Depending on different API calls, the influence on the status of RAM blocks shall be described in addition to the specification items located in chapter 8.3.

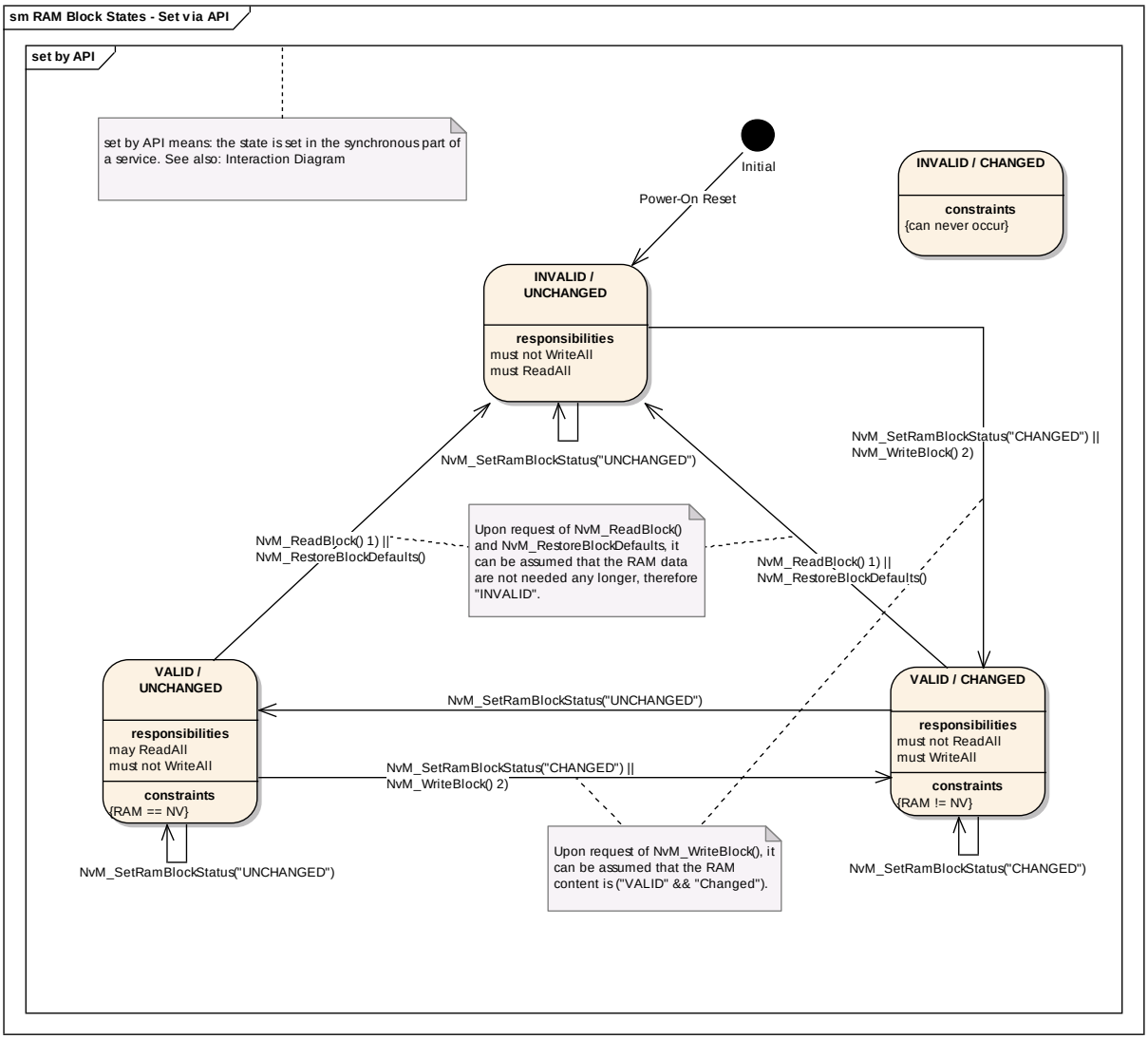


**Figure 8: RAM block states I**

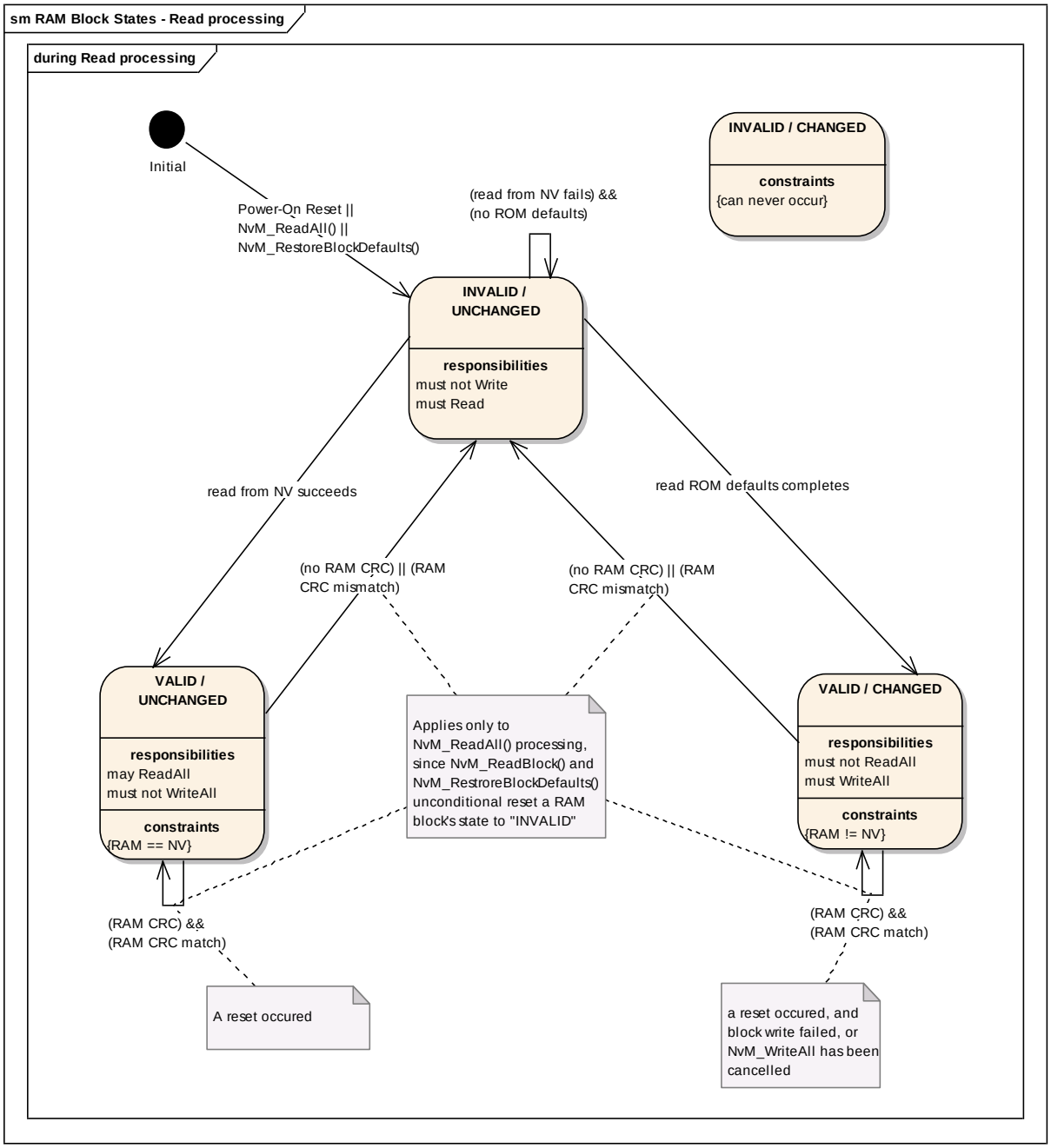


**Figure 9: RAM block states II**

**NVM344:** If the API for modifying the RAM block status has been configured out (via `NvmSetRamBlockStatusApi`) the NvM module shall treat a RAM block or the RAM mirror in the NvM module (in case of explicit synchronization) as valid and modified when writing to it, i.e. during `NvM_WriteAll`, the NvM module shall write each permanent RAM block to NV memory.

**NVM345:** If the API for modifying the RAM block status has been configured out (via `NvmSetRamBlockStatusApi`) the NvM module shall treat a RAM block as invalid

when reading it, i.e. during `NvM_ReadAll`, the NvM module shall copy each NVRAM block to RAM if configured accordingly.

Note: In case of an unsuccessful block read attempt, it is the responsibility of the application to provide valid data before the next write attempt.

#### 7.2.2.16 Communication and synchronization between application and NVRAM manager

To minimize locking/unlocking overhead or the use of other synchronization methods, the communication between applications and the NvM module must follow a strict sequence of steps which is described below. This ensures a reliable communication between applications and the NvM module and avoids data corruption in RAM blocks and a proper synchronization is guaranteed.

This access model assumes that two parties are involved in communication with a RAM block: The application and the NvM module.

If several applications are using the same RAM block it is not the job of the NvM module to ensure the data integrity of the RAM block. In this case, the applications have to synchronize their accesses to the RAM block and have to guarantee that no unsuitable accesses to the RAM block take place during NVRAM operations (details see below).
Especially if several applications are sharing a NVRAM block by using (different) temporary RAM blocks, synchronization between applications becomes more complex and this is not handled by the NvM module, too. In case of using callbacks as notification method, it could happen that e.g. an application gets a notification although the request has not been initiated by this application.

All applications have to adhere to the following rules.

##### 7.2.2.16.1 Write requests (NvM_WriteBlock)

1. The application fills a RAM block with the data that has to be written by the NvM module.

2. The application issues the `NvM_WriteBlock` request which transfers control to the NvM module.

3. From now on the application must not modify the RAM block until success or failure of the request is signaled or derived via polling. In the meantime the contents of the RAM block may be read.

4. An application can use polling to get the status of the request or can be informed via a callback function asynchronously.

5. After completion of the NvM module operation, the RAM block is reusable for modifications.

#### 7.2.2.16.2 Read requests (NvM_ReadBlock)

1. The application provides a RAM block that has to be filled with NVRAM data from the NvM module's side.

2. The application issues the `NvM_ReadBlock` request which transfers control to the NvM module.

3. From now on the application must not read or write to the RAM block until success or failure of the request is signaled or derived via polling.

4. An application can use polling to get the status of the request or can be informed via a callback function.

5. After completion of the NvM module operation, the RAM block is available with new data for use by the application.

#### 7.2.2.16.3 Restore default requests (NvM_RestoreBlockDefaults)

1. The application provides a RAM block, which has to be filled with ROM data from the NvM modules side.

2. The application issues the NvM_RestoreBlockDefaults request which transfers control to the NvM module.

3. From now on the application must not read or write to the RAM block until success or failure of the request is signaled or derived via polling.

4. An application can use polling to get the status of the request or can be informed via a callback function.

5. After completion of the NvM module operation, the RAM block is available with the ROM data for use by the application.

#### 7.2.2.16.4 Multi block read requests (NvM_ReadAll)

This request may be triggered only by the ECU state manager at system startup. This request fills all configured permanent RAM blocks with necessary data for startup.

If the request fails or the request is handled only partially successful, the NVRAM-Manager signals this condition to the DEM and returns an error to the ECU state manager. The DEM and the ECU state manager have to decide about further measures that have to be taken. These steps are beyond the scope of the NvM module and are handled in the specifications of DEM and ECU state manager.

Normal operation:

1. The ECU state manager issues the `NvM_ReadAll`.

2. The ECU state manager can use polling to get the status of the request or can be informed via a callback function.

3. During `NvM_ReadAll`, a single block callback (if configured) will be invoked after having completely processed a NVRAM block. These callbacks enable the RTE to start each SW-C individually.

#### 7.2.2.16.5 Multi block write requests (NvM_WriteAll)

This request must only be triggered by the ECU state manager at shutdown of the system. This request writes the contents of all modified permanent RAM blocks to NV memory. By calling this request only during ECU shutdown, the ECU state manager can ensure that no SW component is able to modify data in the RAM blocks until the end of the operation. These measures are beyond the scope of the NvM module and are handled in the specifications of the ECU state manager.

Normal operation:

1. The ECU state manager issues the `NvM_WriteAll` request which transfers control to the NvM module.

2. The ECU state manager can use polling to get the status of the request or can be informed via a callback function.

#### 7.2.2.16.6 Cancel Operation (NvM_CancelWriteAll)

This request cancels a pending `NvM_WriteAll` request. This is an asynchronous request and can be called to terminate a pending `NvM_WriteAll` request. This request shall only be used by the ECU state manager.

#### 7.2.2.16.7 Modification of administrative blocks

For administrative purposes an administrative block is part of each configured NVRAM block (ref. to ch. 7.1.3.4).

If there is a pending single-block operation for a NVRAM block, the

application is not allowed to call any operation that modifies the administrative block, like `NvM_SetDataIndex`, `NvM_SetBlockProtection`, `SetRamBlockStatus`, until the pending job has finished.

#### 7.2.2.17 Communication and explicit synchronization between application and NVRAM manager

In contrast to the implicit synchronization between the application and the NvM module (see section 7.2.2.16) an optional (i.e. configurable) explicit synchronization mechanism is available. It is realized by a RAM mirror in the NvM module. The data is transferred by the application in both directions via callback routines, called by the NvM module.

Here is a short analysis of this mechanism:

- The advantage is that applications can control their data in a better way. They are responsible for copying consistent data to and from the NvM module's RAM mirror, so they know the point in time. The RAM block is never in an inconsistent state due to concurrent accesses.
- The drawbacks are the additional RAM which needs to have the same size as the largest NVRAM block that uses this mechanism and the necessity of an additional copy between two RAM locations for every operation.

This mechanism especially enables the sharing of NVRAM blocks by different applications, if there is a module that synchronizes these applications and is the owner of the NVRAM block from the NvM module's perspective.

**NVM511:** For every NVRAM block there shall be the possibility to configure the usage of an explicit synchronization mechanism by the parameter NvMBlockUseSyncMechanism.

**NVM512:** The NvM module must not allocate a RAM mirror if no block is configured to use the explicit synchronization mechanism.

**NVM513:** The NvM module shall allocate only one RAM mirror if at least one block is configured to use the explicit synchronization mechanism. This RAM mirror must not exceed the size of the longest NVRAM block configured to use the explicit synchronization mechanism.

**NVM514:** The NvM module shall use the internal mirror as buffer for all operations that read and write the RAM block of those NVRAM blocks with NvMBlockUseSyncMechanism == TRUE. The buffer must not be used for the other NVRAM blocks.

**NVM515:** The NvM module shall call the routine NvMWriteRamBlockToNvM in order to copy the data from the RAM block to the mirror for all NVRAM blocks with

NvMBlockUseSyncMechanism == TRUE. This routine must not be used for the other NVRAM blocks.

**NVM516:** The NvM module shall call the routine NvMReadRamBlockFromNvM in order to copy the data from the mirror to the RAM block for all NVRAM blocks with NvMBlockUseSyncMechanism == TRUE. This routine must not be used for the other NVRAM blocks.

**NVM517:** During a single block request if the routines NvMReadRamBlockFromNvM return E_NOT_OK, then the NvM module shall retry the routine call NvMRepeatMirrorOperations times. Thereafter the single block read job shall set the block specific job result to NVM_REQ_NOT_OK and shall report NVM_E_REQ_FAILED to the DEM.

**NVM839:** In the case the NvMReadRamBlockFromNvM routine returns E_NOT_OK, the NvM module shall retry the routine call in the next call of the NvM_MainFunction.

**NVM579:** During a single block request if the routines NvMWriteRamBlockToNvM return E_NOT_OK, then the NvM module shall retry the routine call NvMRepeatMirrorOperations times. Thereafter the single block write job shall set the block specific job result to NVM_REQ_NOT_OK and shall report NVM_E_REQ_FAILED to the DEM.

**NVM840:** In the case the NvMWriteRamBlockToNvM routine returns E_NOT_OK, the NvM module shall retry the routine call in the next call of the NvM_MainFunction.

**NVM837:** During a multi block request(NvM_WriteAll) if the routines NvMWriteRamBlockToNvM return E_NOT_OK, then the NvM module shall retry the routine call NvMRepeatMirrorOperations times. Thereafter the job of the function NvM_WriteAll shall set the block specific job result to NVM_REQ_NOT_OK and shall report NVM_E_REQ_FAILED to the DEM.

**NVM838:** During a multi block request(NvM_ReadAll) if the routines NvMReadRamBlockFromNvM return E_NOT_OK, then the NvM module shall retry the routine call NvMRepeatMirrorOperations times. Thereafter the job of the function NvM_ReadAll shall set the block specific job result to NVM_REQ_NOT_OK and shall report NVM_E_REQ_FAILED to the DEM.

The following two sections clarify the differences when using the explicit synchronization mechanism, compare to 7.2.2.16.1 and 7.2.2.16.2 .

#### 7.2.2.17.1 Write requests (NvM_WriteBlock)

**NVM705:** Applications have to adhere to the following rules during write request for explicit synchronization between application and NVRAM manager:

1. The application fills a RAM block with the data that has to be written by the NvM module.

2. The application issues the NvM_WriteBlock request.
3. The application might modify the RAM block until the routine NvMWriteRamBlockToNvM is called by the NvM module.
4. If the routine NvMWriteRamBlockToNvM is called by the NvM module, then the application has to provide a consistent copy of the RAM block to the destination requested by the NvM module.
   The application can use the return value E_NOT_OK in order to signal that data was not consistent. The NvM module will accept this NvMRepeatMirrorOperations times and then postpones the request and continues with its next request.
5. Continuation only if data was copied to the NvM module:
6. From now on the application can read and write the RAM block again.
7. An application can use polling to get the status of the request or can be informed via a callback routine asynchronously.
8. Note: The application may combine several write requests to different positions in one RAM block, if NvM_WriteBlock was requested, but not yet processed by the NvM module. The request was not processed, if the callback routine NvMWriteRamBlockToNvM was not called.

#### 7.2.2.17.2 Read requests (NvM_ReadBlock)

**NVM706:** Applications have to adhere to the following rules during read request for explicit synchronization between application and NVRAM manager:

1. The application provides a RAM block that has to be filled with NVRAM data from the NvM module's side.
2. The application issues the NvM_ReadBlock request.
3. The application might modify the RAM block until the routine NvMReadRamBlockFromNvM is called by the NvM module.
4. If the routine NvMReadRamBlockFromNvM is called by the NvM module, then the application copy the data from the destination given by the NvM module to the RAM block.The application can use the return value E_NOT_OK in order to signal that data was not copied. The NvM module will accept this NvMRepeatMirrorOperations times and then postpones the request and continues with its next request.
5. Continuation only if data was copied from the NvM module:
6. Now the application finds the NV block values in the RAM block.
7. The application can use polling to get the status of the request or can be informed via a callback routine.

Note: The application may combine several read requests to different positions in one NV block, if NvM_ReadBlock was requested, but not yet processed by the NvM module. The request was not processed, if the callback routine NvMReadRamBlockFromNvM was not called.
Note: NvM_RestoreBlockDefaults works similarly to NvM_ReadBlock.

#### 7.2.2.17.3 Multi block read requests (NvM_ReadAll)

This request may be triggered only by the ECU state manager at system startup. This request fills all configured permanent RAM blocks with necessary data for startup.

If the request fails or the request is handled only partially successful, the NVRAM-Manager signals this condition to the DEM and returns an error to the ECU state manager. The DEM and the ECU state manager have to decide about further measures that have to be taken. These steps are beyond the scope of the NvM module and are handled in the specifications of DEM and ECU state manager.

Normal operation:

1. The ECU state manager issues the `NvM_ReadAll`.

2. The ECU state manager can use polling to get the status of the request or can be informed via a callback function.

3. During `NvM_ReadAll` job, if a synchronization callback (NvM_ReadRamBlockFromNvm) is configured for a block it will be called by the NvM module. In this callback the application shall copy the data from the destination given by the NvM module to the RAM block.The application can use the return value E_NOT_OK in order to signal that data was not copied. The NvM module will accept this NvMRepeatMirrorOperations times and then report the read operation as failed.

4. Now the application finds the NV block values in the RAM block if the read operation was successful.

5. During `NvM_ReadAll`, a single block callback (if configured) will be invoked after having completely processed a NVRAM block. These callbacks enable the RTE to start each SW-C individually.

#### 7.2.2.17.4 Multi block write requests (NvM_WriteAll)

This request must only be triggered by the ECU state manager at shutdown of the system. This request writes the contents of all modified permanent RAM blocks to NV memory. By calling this request only during ECU shutdown, the ECU state manager can ensure that no SW component is able to modify data in the RAM blocks until the end of the operation. These measures are beyond the scope of the NvM module and are handled in the specifications of the ECU state manager.

Normal operation:

1. The ECU state manager issues the `NvM_WriteAll` request which transfers control to the NvM module.

2. During `NvM_WriteAll` job, if a synchronization callback (NvM_WriteRamBlockToNvM) is configured for a block it will be called by the NvM module. In this callback the application has to provide a consistent copy of the RAM block to the destination requested by the NvM module. The application can use the return value E_NOT_OK in order to signal that data was not consistent. The NvM module will accept this NvMRepeatMirrorOperations times and then report the write operation as failed.

3. Now the application can read and write the RAM block again.

4. The ECU state manager can use polling to get the status of the request or can be informed via a callback function.

#### 7.2.2.18 Normal and extended runtime preparation of NVRAM blocks

This subchapter is supposed to provide a short summary of normal and extended runtime preparation of NVRAM blocks. The detailed behavior regarding the handling of NVRAM blocks during start-up is specified in chapter 8.3.3.1.

Depending on the two configuration parameters NvmDynamicConfiguration and NvmResistantToChangedSw the NVRAM Manager shall behave in different ways during start-up, i.e. while processing the request NvM_ReadAll().
If NvmDynamicConfiguration is set to FALSE, the NVRAM Manager shall ignore the stored configuration ID and continue with the normal runtime preparation of NVRAM blocks. In this case the RAM block shall be checked for its validity. If the RAM block content is detected to be invalid the NV block shall be checked for its validity. A NV block which is detected to be valid shall be copied to its assigned RAM block. If an invalid NV Block is detected default data shall be loaded.
If NvmDynamicConfiguration is set to TRUE and a configuration ID mismatch is detected, the extended runtime preparation shall be performed for those NVRAM blocks which are configured with NvmResistantToChangedSw(FALSE). In this case default data shall be loaded independent of the validity of an assigned RAM or NV block.

## 7.3 Error classification

**NVM186:** Values for production code Event Ids are assigned externally by the configuration of the Dem. They are published in the file `Dem_IntErrId.h` and included via `Dem.h`.

**NVM739:** If not specified in a special case differently: For all production errors reported to the DEM the EventStatus shall be set to DEM_EVENT_STATUS_FAILED.

**NVM187:** Development error values are of type uint8.

**NVM023:** The following errors and exceptions shall be detectable by the NvM module depending on its build version (development/production mode).

| *Type or error* | *Relevance* | *Related error code* | *Value[hex]* |
|---|---|---|---|
| API requests called with wrong parameter | Development | NVM_E_PARAM_BLOCK_ID<br>NVM_E_PARAM_BLOCK_TYPE<br>NVM_E_PARAM_BLOCK_DATA_IDX<br>NVM_E_PARAM_ADDRESS<br>NVM_E_PARAM_DATA | 0x0A<br>0x0B<br>0x0C<br>0x0D<br>0x0E |
| NVRAM manager is still not initialized | Development | NVM_E_NOT_INITIALIZED | 0x14 |
| API read/write/control request failed because a block with the same ID is already listed or currently in progress | Development | NVM_E_BLOCK_PENDING | 0x15 |
| NVRAM manager job queue overflow occurred | Development | NVM_E_LIST_OVERFLOW | 0x16 |
| A write attempt to a write protected NVRAM block was requested. | Development | NVM_E_NV_WRITE_PROTECTED | 0x17 |
| The service is not possible with this block configuration. | Development | NVM_E_BLOCK_CONFIG | 0x18 |
| API request integrity failed | Production | NVM_E_INTEGRITY_FAILED | Assigned by DEM |
| API request failed | Production | NVM_E_REQ_FAILED | Assigned by DEM |

**NVM024:** Additional errors that are detected because of specific implementation and/or specific hardware properties shall be added in the NvM module implementation specification. The classification and enumeration shall be compatible to the errors listed above [NVM023].

## 7.4 Error detection

**NVM025:** The detection of development errors is configurable (*ON* / *OFF*) at pre-compile time. The switch `NvmDevErrorDetect` (see chapter 10) shall activate or deactivate the detection of all development errors.

**NVM188:** If the `NvmDevErrorDetect` switch is enabled API parameter checking is enabled. The detailed description of the detected errors can be found in chapter 7.3.

**NVM189:** The detection of production code errors cannot be switched off.

**NVM027:** If development error detection is enabled for this module [NVM028], the following table specifies which DET error values shall be reported for each API call:

| API call | Error condition | DET related error value |
|---|---|---|
| NvM_Init | -- | -- |
| NvM_SetDataIndex | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| | NVRAM block identifier is already queued or currently in progress | NVM_E_BLOCK_PENDING |
| | DataIndex parameter exceeds the total number of configured datasets NVM444, NVM445 | NVM_E_PARAM_BLOCK_DATA_IDX |
| | The request is not possible in conjunction with the configured block management type. | NVM_E_PARAM_BLOCK_TYPE |
| | The passed BlockID is out of range | NVM_E_PARAM_BLOCK_ID |
| NvM_GetDataIndex | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| | The request is not possible in conjunction with the configured block management type. | NVM_E_PARAM_BLOCK_TYPE |
| | The passed BlockID is out of range | NVM_E_PARAM_BLOCK_ID |
| | A NULL pointer is passed via the parameter DataIndexPtr. | NVM_E_PARAM_DATA |
| NvM_SetBlockProtection | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| | NVRAM block identifier is already queued or currently in progress | NVM_E_BLOCK_PENDING |
| | The NVRAM block is configured with NvmWriteBlockOnce = TRUE | NVM_E_BLOCK_CONFIG |
| | The passed BlockID is out of range | NVM_E_PARAM_BLOCK_ID |
| NvM_GetErrorStatus | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| | The passed BlockID is out of range | NVM_E_PARAM_BLOCK_ID |
| | A NULL pointer is passed via the parameter RequestResultPtr. | NVM_E_PARAM_DATA |
| NvM_GetVersionInfo | A NULL pointer is passed via the parameter versioninfo. | NVM_E_PARAM_DATA |
| NvM_ReadBlock | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| | NVRAM block identifier is already queued or currently in progress | NVM_E_BLOCK_PENDING |
| | No permanent RAM block is configured and a NULL pointer is passed via the parameter NvM_DstPtr. | NVM_E_PARAM_ADDRESS |
| | A job queue overflow occured | NVM_E_LIST_OVERFLOW |
| | The passed BlockID is out of range | NVM_E_PARAM_BLOCK_ID |
| NvM_WriteBlock | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| | NVRAM block identifier is already queued or currently in progress | NVM_E_BLOCK_PENDING |
| | A job queue overflow occured | NVM_E_LIST_OVERFLOW |
| | No permanent RAM block is configured and a NULL pointer is passed via the parameter NvM_SrcPtr | NVM_E_PARAM_ADDRESS |
| | A write protected NVRAM block is referenced by the passed BlockID. | NVM_E_NV_WRITE_PROTECTED |

| | | |
|---|---|---|
| | The passed BlockID is out of range | NVM_E_PARAM_BLOCK_ID |
| NvM_RestoreBlockDefaults | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| | NVRAM block identifier is already queued or currently in progress | NVM_E_BLOCK_PENDING |
| | A job queue overflow occured | NVM_E_LIST_OVERFLOW |
| | Default data is not available/configured for the referenced NVRAM block. | NVM_E_BLOCK_CONFIG |
| | No permanent RAM block is configured and a NULL pointer is passed via the parameter NvM_DstPtr | NVM_E_PARAM_ADDRESS |
| | The passed BlockID is out of range | NVM_E_PARAM_BLOCK_ID |
| NvM_EraseNvBlock | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| | NVRAM block identifier is already queued or currently in progress | NVM_E_BLOCK_PENDING |
| | A job queue overflow occured | NVM_E_LIST_OVERFLOW |
| | A write protected NVRAM block is referenced by the passed BlockID. | NVM_E_NV_WRITE_PROTECTED |
| | The passed BlockID is out of range | NVM_E_PARAM_BLOCK_ID |
| | The NVRAM block has not immediate priority | NVM_E_BLOCK_CONFIG |
| NvM_CancelWriteAll | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| NvM_InvalidateNvBlock | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| | NVRAM block identifier is already queued or currently in progress | NVM_E_BLOCK_PENDING |
| | A job queue overflow occured | NVM_E_LIST_OVERFLOW |
| | A write protected NVRAM block is referenced by the passed BlockID. | NVM_E_NV_WRITE_PROTECTED |
| | The passed BlockID is out of range | NVM_E_PARAM_BLOCK_ID |
| NvM_SetRamBlockStatus | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| | NVRAM block identifier is already queued or currently in progress | NVM_E_BLOCK_PENDING |
| | The passed BlockID is out of range | NVM_E_PARAM_BLOCK_ID |
| NvM_ReadAll | NVM not yet initialized | NVM_E_NOT_INITIALIZED |
| NvM_WriteAll | NVM not yet initialized | NVM_E_NOT_INITIALIZED |

## 7.5 Error notification

**NVM026:** Production errors shall be reported to the Diagnostic Event Manager.

**NVM191:** Detected development errors shall be reported to the `Det_ReportError` service of the Development Error Tracer (DET) if the pre-processor switch `PWM_DEV_ERROR_DETECT` is set (see chapter 10).

## 7.6 Version check

**NVM089:** NvM.c shall check if the correct version of NvM.h is included. This shall be done by a preprocessor check of the version number `NVM_SW_MAJOR_VERSION` [NVM022].

# 8 API specification

## 8.1 Imported types

In this chapter all types included from the following files are listed:

**NVM446:**

| *Module* | *Imported Type* |
|---|---|
| Dem | Dem_EventIdType |
| MemIf | MemIf_JobResultType |
| | MemIf_ModeType |
| | MemIf_StatusType |
| Std_Types | Std_ReturnType |
| | Std_VersionInfoType |

## 8.2 Type definitions

### 8.2.1 NvM_RequestResultType

**NVM083:** The type NvM_RequestResultType is an asynchronous request result, which will be returned by the API service `NvM_GetErrorStatus` [NVM015].

**NVM470:**

| | | | |
|---|---|---|---|
| ***Name:*** | `NvM_RequestResultType` | | |
| ***Type:*** | `uint8` | | |
| ***Range:*** | `NVM_REQ_OK` | 0 | The last asynchronous read/write/control request has been finished successfully. This shall be the default value after reset. This status shall have the value 0. |
| | `NVM_REQ_NOT_OK` | 1 | The last asynchronous read/write/control request has been finished unsuccessfully. |
| | `NVM_REQ_PENDING` | 2 | An asynchronous read/write/control request is currently pending. |
| | `NVM_REQ_INTEGRITY_FAILED` | 3 | The result of the last asynchronous request<br><br>NvM_ReadBlock or NvM_ReadAll<br><br>is a data integrity failure.<br><br>Note:<br>In case of<br><br>NvM_ReadBlock<br><br>the content of the RAM block has changed but |

| | | | |
|---|---|---|---|
| | | | has become invalid. The application is responsible to renew and validate the RAM block content. |
| | NVM_REQ_BLOCK_SKIPPED | 4 | The referenced block was skipped during execution of NvM_ReadAll or NvM_WriteAll, e.g. Dataset NVRAM blocks (NvM_ReadAll) or NVRAM blocks without a permanently configured RAM block. |
| | NVM_REQ_NV_INVALIDATED | 5 | The referenced NV block is invalidated. |
| | NVM_REQ_CANCELLED | 6 | The multi block request NvM_WriteAll was cancelled by calling NvM_CancelWriteAll. |
| ***Description:*** | This is an asynchronous request result returned by the API service NvM_GetErrorStatus. The availability of an asynchronous request result can be additionally signaled via a callback function. | | |

### 8.2.2 NvM_BlockIdType

**NVM471:**

| | | | |
|---|---|---|---|
| ***Name:*** | NvM_BlockIdType | | |
| ***Type:*** | uint16 | | |
| ***Range:*** | 0..2^(16-NVM_DATASET_SELECTION_BITS)-1 | -- | -- |
| ***Description:*** | Identification of a NVRAM block via a unique block identifier.<br><br>Reserved NVRAM block IDs:<br>0 -> to derive multi block request results via NvM_GetErrorStatus<br>1 -> redundant NVRAM block which holds the configuration ID | | |

**NVM475:** The NVRAM block IDs shall be in a sequential order, i.e. the NVRAM manager does not need to be capable of handling non-sequential NVRAM block IDs.

Example: If 50 NvM block have to be configured then their IDs are expected to be configured from 2 until 51 because block ID 0 and 1 are reserved for NvM intenal use. So the sequential order will start with the ID 0 and increase one by one until 51, however only the blocks with IDs from 2 to 51 can and will be configured.

## 8.3 Function definitions

### 8.3.1 Synchronous requests

#### 8.3.1.1 NvM_Init

**NVM447:**

| | |
|---|---|
| ***Service name:*** | NvM_Init |
| ***Syntax:*** | `void NvM_Init(`<br><br>`)` |
| ***Service ID[hex]:*** | 0x00 |
| ***Sync/Async:*** | Synchronous |
| ***Reentrancy:*** | Non Reentrant |
| ***Parameters (in):*** | None |
| ***Parameters (inout):*** | None |
| ***Parameters (out):*** | None |
| ***Return value:*** | None |
| ***Description:*** | Service for resetting all internal variables. |

**NVM399:** The function `NvM_Init` shall reset all internal variables, e.g. the queues, request flags, state machines, to their initial values. It shall signal “INIT DONE” internally, e.g. to enable job processing and queue management.

**NVM400:** The function NvM_Init shall not modify the permanent RAM block contents or call explicit synchronization callback, as this shall be done on `NvM_ReadAll`.

**NVM192:** The function `NvM_Init` shall set the dataset index of all NVRAM blocks of type `NVM_BLOCK_DATASET` to zero.

**NVM193:** The function `NvM_Init` shall not initialize other modules (it is assumed that the underlying layers are already initialized).

The function `NvM_Init` is affected by the common [NVM028] and published [NVM022] configuration parameter.

Hint:
The time consuming NVRAM block initialization and setup according to the block descriptor [NVM061] shall be done by the `NvM_ReadAll` request [NVM008].

#### 8.3.1.2 NvM_SetDataIndex

**NVM448:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_SetDataIndex | |
| ***Syntax:*** | `void NvM_SetDataIndex(`<br>`  NvM_BlockIdType BlockId,`<br>`  uint8 DataIndex`<br>`)` | |
| ***Service ID[hex]:*** | 0x01 | |
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Reentrant | |
| ***Parameters (in):*** | BlockId | The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block. |
| | DataIndex | Index position (association) of a NV/ROM block. |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | None | |
| ***Return value:*** | None | |
| ***Description:*** | Service for setting the DataIndex of a dataset NVRAM block. | |

**NVM014:** The function `NvM_SetDataIndex` shall set the index to access a certain dataset of a NVRAM block (with/without ROM blocks).

**NVM263:** The function `NvM_SetDataIndex` shall leave the content of the corresponding RAM block unmodified.

**NVM264:** The NvM module's environment shall use the function `NvM_SetDataIndex` in conjunction with dataset NVRAM block management types. The function `NvM_SetDataIndex` shall be able to be used in conjunction with all other block management types but without any effect in production mode.

Regarding error detection, the requirement NVM027 is applicable to the function `NvM_SetDataIndex`.

**NVM401:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_SetDataIndex`.

Hint:
NVRAM common block configuration parameters [NVM028], block management types [NVM061] and one configured NVRAM block descriptor needed [NVM062].

#### 8.3.1.3 NvM_GetDataIndex

**NVM449:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_GetDataIndex | |
| ***Syntax:*** | `void NvM_GetDataIndex(`<br>`  NvM_BlockIdType BlockId,`<br>`  uint8* DataIndexPtr`<br>`)` | |
| ***Service ID[hex]:*** | 0x02 | |
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Reentrant | |
| ***Parameters (in):*** | BlockId | The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block. |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | DataIndexPtr | Pointer to where to store the current dataset index (0..255) |
| ***Return value:*** | None | |
| ***Description:*** | Service for getting the currently set DataIndex of a dataset NVRAM block | |

**NVM021:** The function `NvM_GetDataIndex` shall get the current index (association) of a dataset NVRAM block (with/without ROM blocks).

**NVM265:** The function `NvM_GetDataIndex` shall be able to be used in conjunction with all other block management types than dataset block management type is possible but without any effect in production mode. In this case, the function shall return zero, i.e. the function shall set the pointer DataIndexPtr to zero.

Regarding error detection, the requirement NVM027 is applicable to the function `NvM_GetDataIndex`.

**NVM402:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_GetDataIndex`.

Hint:
NVRAM common block configuration parameters [NVM028], block management types [NVM061] and one configured NVRAM block descriptor needed [NVM062].

#### 8.3.1.4 NvM_SetBlockProtection

**NVM450:**

| | |
|---|---|
| ***Service name:*** | NvM_SetBlockProtection |
| ***Syntax:*** | `void NvM_SetBlockProtection(`<br>`  NvM_BlockIdType BlockId,`<br>`  boolean ProtectionEnabled`<br>`)` |
| ***Service ID[hex]:*** | 0x03 |

| | | |
|---|---|---|
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Reentrant | |
| ***Parameters (in):*** | BlockId | The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block. |
| | ProtectionEnabled | TRUE: Write protection shall be enabled<br>FALSE: Write protection shall be disabled |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | None | |
| ***Return value:*** | None | |
| ***Description:*** | Service for setting/resetting the write protection for a NV block. | |

**NVM016:** The function `NvM_SetBlockProtection` shall set/reset the write protection for the corresponding NV block by setting the write protection attribute in the administrative part of the corresponding NVRAM block.

**NVM276:** The function `NvM_SetBlockProtection` shall not change the write protection of NV blocks with `NvmWriteBlockOnce == TRUE`.

Regarding error detection, the requirement NVM027 is applicable to the function `NvM_SetBlockProtection`.

**NVM403:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_SetBlockProtection`.

Hint:
NVRAM common block configuration parameters [NVM028], block management types [NVM061] and one configured NVRAM block descriptor needed [NVM062].

### 8.3.1.5 NvM_GetErrorStatus

**NVM451:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_GetErrorStatus | |
| ***Syntax:*** | `void NvM_GetErrorStatus(`<br>`  NvM_BlockIdType BlockId,`<br>`  uint8* RequestResultPtr`<br>`)` | |
| ***Service ID[hex]:*** | 0x04 | |
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Reentrant | |
| ***Parameters (in):*** | BlockId | The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block. |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | RequestResultPtr | Pointer to where to store the request result. See NvM_RequestResultType . |
| ***Return value:*** | None | |

| | |
|---|---|
| ***Description:*** | Service to read the block dependent error/status information. |

**NVM015:** The function `NvM_GetErrorStatus` shall read the block dependent error/status information in the administrative part of a NVRAM block.

The status/error information of a NVRAM block shall be set by a former or current asynchronous request.

Regarding error detection, the requirement NVM027 is applicable to the function NvM_GetErrorStatus.

**NVM404:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_GetErrorStatus`.

NVRAM common block configuration parameters [NVM028], block management types [NVM061] and one configured NVRAM block descriptor are needed in the configuration with respect to the function `NvM_GetErrorStatus` [NVM062].

#### 8.3.1.6 NvM_GetVersionInfo

**NVM452:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_GetVersionInfo | |
| ***Syntax:*** | `void NvM_GetVersionInfo(`<br>`  Std_VersionInfoType* versioninfo`<br>`)` | |
| ***Service ID[hex]:*** | 0x0f | |
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Non Reentrant | |
| ***Parameters (in):*** | None | |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | versioninfo | Pointer to where to store the version information of this module. |
| ***Return value:*** | None | |
| ***Description:*** | Service to get the version information of the NvM module. | |

**NVM285:** The function `NvM_GetVersionInfo` shall return the version information of this module. The version information includes:

- Module Id
- Vendor Id
- Vendor specific version numbers (BSW00407).

**NVM286:** The function `NvM_GetVersionInfo` shall be pre-compile time configurable On/Off by the configuration parameter: `NvmVersionInfoApi`

Hint:

If source code for caller and callee of the function `NvM_GetVersionInfo` is available, the function should be realized as a macro. The macro should be defined in the modules header file.

The function `NvM_GetVersionInfo` is affected by the common block configuration parameter [NVM028].

#### 8.3.1.7 NvM_SetRamBlockStatus

**NVM453:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_SetRamBlockStatus | |
| ***Syntax:*** | `void NvM_SetRamBlockStatus( NvM_BlockIdType BlockId, boolean BlockChanged )` | |
| ***Service ID[hex]:*** | 0x05 | |
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Reentrant | |
| ***Parameters (in):*** | BlockId | The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block. |
| | BlockChanged | TRUE: Validate the RAM block and mark block as changed.<br>FALSE: Invalidate the RAM block and mark block as unchanged. |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | None | |
| ***Return value:*** | None | |
| ***Description:*** | Service for setting the RAM block status of an NVRAM block. | |

**NVM240:** The function `NvM_SetRamBlockStatus` shall only work on NVRAM blocks with a permanently configured RAM block and shall have no effect to other NVRAM blocks.

**NVM241:** The function `NvM_SetRamBlockStatus` shall assume that a changed permanent RAM block or the content of the RAM mirror in the NvM module ( in case of explicit synchronization) is valid (basic assumption).

**NVM405:** When the “BlockChanged” parameter passed to the function `NvM_SetRamBlockStatus` is FALSE the corresponding RAM block is either invalid or unchanged (or both).

**NVM406:** When the “BlockChanged” parameter passed to the function `NvM_SetRamBlockStatus` is TRUE, the corresponding permanent RAM block or the content of the RAM mirror in the NvM module ( in case of explicit synchronization) is valid and changed.

**NVM121:** The function `NvM_SetRamBlockStatus` shall request the recalculation of CRC in the background, i.e. the CRC recalculation shall be processed by the NvM_MainFunction, if the given "BlockChanged" parameter is TRUE and CRC calculation in RAM is configured (i.e. `NvmCalcRamBlockCrc == TRUE`).

Hint:
In some cases, a permanent RAM block cannot be validated neither by a reload of its NV data, nor by a load of its ROM data during the execution of a `NvM_ReadAll` command (startup). The application is responsible to fill in proper data to the RAM block and to validate the block via the function `NvM_SetRamBlockStatus` before this RAM block can be written to its corresponding NV block by `NvM_WriteAll`.

It is expected that the function `NvM_SetRamBlockStatus` will be called frequently for NVRAM blocks which are configured to be protected in RAM via CRC. Otherwise this function only needs to be called once to mark a block as "changed" and to be processed during `NvM_WriteAll`.

Regarding error detection, the requirement NVM027 is applicable to the function NvM_SetRamBlockStatus.

**NVM407:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_SetRamBlockStatus`.

**NVM408**: The NvM module shall provide the function `NvM_SetRamBlockStatus` only if it is configured via `NvmSetRamBlockStatusApi` [NVM028].

NVRAM common configuration parameters [NVM028], block management types [NVM061] and one configured NVRAM block descriptor are needed in the configuration with respect to the function `NvM_SetRamBlockStatus` [NVM062].

### 8.3.2 Asynchronous single block requests

#### 8.3.2.1 NvM_ReadBlock

**NVM454:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_ReadBlock | |
| ***Syntax:*** | `Std_ReturnType NvM_ReadBlock( NvM_BlockIdType BlockId, uint8* NvM_DstPtr )` | |
| ***Service ID[hex]:*** | 0x06 | |
| ***Sync/Async:*** | Asynchronous | |
| ***Reentrancy:*** | Reentrant | |
| ***Parameters (in):*** | BlockId | The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block. |
| ***Parameters*** | None | |

| | | |
|---|---|---|
| ***(inout):*** | | |
| ***Parameters (out):*** | NvM_DstPtr | Pointer to the RAM data block. |
| ***Return value:*** | Std_ReturnType | E_OK: request has been accepted<br>E_NOT_OK: request has not been accepted |
| ***Description:*** | Service to copy the data of the NV block to its corresponding RAM block. | |

**NVM010:** The job of the function `NvM_ReadBlock` shall copy the data of the NV block to the corresponding RAM block.

**NVM195:** The function `NvM_ReadBlock` shall take over the given parameters, queue the read request in the job queue and return.

**NVM196:** If the function is provided with a valid RAM block address, it is used. If a NULL pointer is provided and if a permanent block or an NvMBlockUseSyncMechanism is specified the permanent block of the APIs shall be used. Otherwise a DET-Parameter error shall be emitted.

**NVM278:** The job of the function `NvM_ReadBlock` shall provide the possibility to copy NV data to a temporary RAM block although the NVRAM block is configured with a permanent RAM block or explicit synchronization callbacks. In this case, the parameter NvM_DstPtr must be unequal to the NULL pointer.

**NVM198:** The function `NvM_ReadBlock` shall invalidate a permanent RAM block immediately when the block is successfully enqueued or the job processing starts, i.e. copying data from NV memory or ROM to RAM. If the block has a synchronization callback (NvM_ReadRamBlockFromNvm) configured the invalidation will be done just before NvMReadRamBlockFromNvM is called.

[**NVM199:** The job of the function `NvM_ReadBlock` shall initiate a read attempt on the second NV block if the passed BlockId references a NVRAM block of type `NVM_BLOCK_REDUNDANT` and the read attempt on the first NV block fails.

**NVM340:** In case of NVRAM block management type `NVM_BLOCK_DATASET`, the job of the function `NvM_ReadBlock` shall copy only that NV block to the corresponding RAM block which is selected via the data index in the administrative block.

**NVM355:** The job of the function `NvM_ReadBlock` shall not copy the NV block to the corresponding RAM block if the NVRAM block management type is `NVM_BLOCK_DATASET` and the NV block selected by the dataset index is invalidate or inconsistent.

**NVM354:** The job of the function `NvM_ReadBlock` shall copy the ROM block to RAM and set the job result to `NVM_REQ_OK` if the NVRAM block management type is `NVM_BLOCK_DATASET` and the dataset index points at a ROM block.

**NVM200:** The job of the function `NvM_ReadBlock` shall set the RAM block to valid and assume it to be unchanged after a successful copy process of the NV block to RAM.

**NVM366:** The job of the function `NvM_ReadBlock` shall set the RAM block to valid and assume it to be changed if the default values are copied to the RAM successfully.

**NVM206:** The job of the function `NvM_ReadBlock` shall set the job result to `NVM_REQ_OK` if the NV block was copied successfully from NV memory to RAM.

**NVM341:** The job of the function `NvM_ReadBlock` shall set the request result to `NVM_REQ_NV_INVALIDATED` and shall report no error to the DEM if the MemIf reports `MEMIF_BLOCK_INVALID`.

**NVM358:** The job of the function `NvM_ReadBlock` shall set the request result to `NVM_REQ_INTEGRITY_FAILED` and report `NVM_E_INTEGRITY_FAILED` to the DEM if the MemIf reports `MEMIF_BLOCK_INCONSISTENT`.

**NVM359:** The job of the function `NvM_ReadBlock` shall set the request result to `NVM_REQ_NOT_OK` and report `NVM_E_REQ_FAILED` to the DEM if the MemIf reports `MEMIF_JOB_FAILED`.

**NVM279:** The job of the function `NvM_ReadBlock` shall set the job result to `NVM_REQ_OK` and report no error to the DEM if the block management type of the given NVRAM block is `NVM_BLOCK_REDUNDANT` and one of the NV blocks was copied successfully from NV memory to RAM.

**NVM316:** The job of the function `NvM_ReadBlock` shall mark every NVRAM block that is not detected by underlying SW as being invalidated and that has been configured with `NvmWriteBlockOnce == TRUE` as write protected.

Hint: This write protection must not be cleared by `NvM_SetBlockProtection`.

**NVM317:** The job of the function `NvM_ReadBlock` shall invalidate a NVRAM block of management type redundant if both NV blocks have been invalidated.

**NVM201:** The job of the function `NvM_ReadBlock` shall request a CRC recalculation over the RAM block data after the copy process [NVM180] if the NV block is configured with CRC.

**NVM202:** The job of the function `NvM_ReadBlock` shall load the default values according to processing of `NvM_RestoreBlockDefaults` if the recalculated CRC is not equal to the CRC stored in NV memory or the read request passed to the underlying layer fails. If there are no default values available, the RAM blocks shall remain invalid.

**NVM203:** The job of the function `NvM_ReadBlock` shall report `NVM_E_INTEGRITY_FAILED` to the DEM if a CRC mismatch occurs.

**NVM204:** The job of the function `NvM_ReadBlock` shall set the job result `NVM_REQ_INTEGRITY_FAILED` if a CRC mismatch occurs.

Regarding error detection, the requirement NVM027 is applicable to the function NvM_ReadBlock.

**NVM409:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_ReadBlock`.

NVRAM common block configuration parameters [NVM028], block management types [NVM061] and one configured NVRAM block descriptor are needed for configuration with respected to the function `NvM_ReadBlock` [NVM062].

#### 8.3.2.2 NvM_WriteBlock

**NVM455:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_WriteBlock | |
| ***Syntax:*** | `Std_ReturnType NvM_WriteBlock(`<br>`  NvM_BlockIdType BlockId,`<br>`  const uint8* NvM_SrcPtr`<br>`)` | |
| ***Service ID[hex]:*** | 0x07 | |
| ***Sync/Async:*** | Asynchronous | |
| ***Reentrancy:*** | Reentrant | |
| ***Parameters (in):*** | BlockId | The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block. |
| | NvM_SrcPtr | Pointer to the RAM data block. |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | None | |
| ***Return value:*** | Std_ReturnType | E_OK: request has been accepted<br>E_NOT_OK: request has not been accepted |
| ***Description:*** | Service to copy the data of the RAM block to its corresponding NV block. | |

**NVM410:** The job of the function `NvM_WriteBlock` shall copy the data of the RAM block to its corresponding NV block.

**NVM411:** The function `NvM_WriteBlock` shall test the write protection attribute of the NV block in the administrative part of the corresponding RAM block.

**NVM217:** The function NvM_WriteBlock shall return with E_NOT_OK, if a write protected NVRAM block is referenced by the passed BlockId parameter.

**NVM208:** The function NvM_WriteBlock shall take over the given parameters, queue the write request in the job queue and return.

**NVM209:** The function NvM_WriteBlock shall check the NVRAM block protection when the request is enqueued but not again before the request is executed.

**NVM300:** The function NvM_WriteBlock shall cancel a pending job immediately in a destructive way if the passed BlockId references a NVRAM block configured to have immediate priority. The immediate job shall be the next active job to be processed.

**NVM210:** If the function is provided with a valid RAM block address, it is used. If a NULL pointer is provided and if a permanent block or an NvMBlockUseSyncMechanism is specified the permanent block of the APIs shall be used. Otherwise a DET-Parameter error shall be emitted.

**NVM280:** The job of the function NvM_WriteBlock shall provide the possibility to copy a temporary RAM block to a NV block although the NVRAM block is configured with a permanent RAM block or explicit synchronization callbacks. In this case, the parameter NvM_SrcPtr must be unequal to a NULL pointer.

**NVM212:** The job of the function NvM_WriteBlock shall request a CRC recalculation before the RAM block will be copied to NV memory if the NV block is configured with CRC [NVM180].

**NVM745:** The job of the function NvM_WriteBlock shall copy the data content of the RAM block to both corresponding NV blocks if the NVRAM block management type of the processed NVRAM block is NVM_BLOCK_REDUNDANT.

**NVM746:** If the processed NVRAM block is of type NVM_BLOCK_REDUNDANT the job of the function NvM_WriteBlock shall start to copy the data of the RAM block to NV block which has not been read during the job started by NvM_ReadBlock or NvM_ReadAll then continue to copy the other NV block.

**NVM338:** The job of the function NvM_WriteBlock shall copy the RAM block to the corresponding NV block which is selected via the data index in the administrative block if the NVRAM block management type of the given NVRAM block is NVM_BLOCK_DATASET.

**NVM303:** The job of the function NvM_WriteBlock shall assume a referenced permanent RAM block or the RAM mirror in the NvM module in case of explicit synchronization to be valid when the request is passed to the NvM module. If the permanent RAM block is still in an invalid state, the function NvM_WriteBlock shall

validate it automatically before copying the RAM block contents to NV memory or before calling explicit synchronization callback (NvM_WriteRamBlockToNvm).

**NVM213:** The job of the function `NvM_WriteBlock` shall check the number of write retries using a write retry counter to avoid infinite loops. Each negative result reported by the memory interface shall be followed by an increment of the retry counter. In case of a retry counter overrun, the job of the function `NvM_WriteBlock` shall set the job result to `NVM_REQ_NOT_OK` and report `NVM_E_REQ_FAILED` to the DEM.

**NVM216:** The configuration parameter `NVM_MAX_NO_OF_WRITE_RETRIES` [NVM028] shall prescribe the maximum number of write retries for the job of the function `NvM_WriteBlock` when RAM block data cannot be written successfully to the corresponding NV block.

**NVM472:** In case of a RAM block is successfully copied to NV memory the RAM block state shall be set to "valid/unmodified" afterwards.

**NVM284:** The job of the function `NvM_WriteBlock` shall set `NVM_REQ_OK` as job result if the passed BlockId references a NVRAM block of type `NVM_BLOCK_REDUNDANT` and at least one of the NV blocks has been written successfully.

**NVM328:** The job of the function `NvM_WriteBlock` shall set the write protection flag in the administrative block immediately if the NVRAM block is configured with `NvmWriteBlockOnce == TRUE` and the data has been written successfully to the NV block.

Regarding error detection, the requirement NVM027 is applicable to the function NvM_WriteBlock.

**NVM412:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_WriteBlock`.

Hint:
To avoid the situation that in case of redundant NVRAM blocks two different NV blocks are containing different but valid data at the same time, each client of the function `NvM_WriteBlock` may call `NvM_InvalidateNvBlock` in advance.

NVRAM common block configuration parameters [NVM028], block management types [NVM061] and one configured NVRAM block descriptor are needed in the configuration with respect to the function `NvM_WriteBlock` [NVM062].

#### 8.3.2.3 NvM_RestoreBlockDefaults

**NVM456:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_RestoreBlockDefaults | |
| ***Syntax:*** | `Std_ReturnType NvM_RestoreBlockDefaults( NvM_BlockIdType BlockId, uint8* NvM_DestPtr )` | |
| ***Service ID[hex]:*** | 0x08 | |
| ***Sync/Async:*** | Asynchronous | |
| ***Reentrancy:*** | Non Reentrant | |
| ***Parameters (in):*** | BlockId | The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block. |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | NvM_DestPtr | Pointer to the RAM data block. |
| ***Return value:*** | Std_ReturnType | E_OK: request has been accepted<br>E_NOT_OK: request has not been accepted |
| ***Description:*** | Service to restore the default data to its corresponding RAM block. | |

**NVM012:** The job of the function `NvM_RestoreBlockDefaults` shall restore the default data to its corresponding RAM block.

**NVM224:** The function `NvM_RestoreBlockDefaults` shall take over the given parameters, queue the request in the job queue and return.

**NVM267:** The job of the function `NvM_RestoreBlockDefaults` shall load the default data from a ROM block if a ROM block is configured.

**NVM266:** The NvM module's environment shall call the function `NvM_RestoreBlockDefaults` to obtain the default data if no ROM block is configured for a NVRAM block and an application callback routine is configured via the parameter `NvmInitBlockCallback`.

**NVM353:** The function `NvM_RestoreBlockDefaults` shall return with `E_NOT_OK` if the block management type of the given NVRAM block is `NVM_BLOCK_DATASET`, at least one ROM block is configured and the data index points at a NV block.

**NVM435:** If the function is provided with a valid RAM block address, it is used. If a NULL pointer is provided and if a permanent block or an NvMBlockUseSyncMechanism is specified the permanent block of the APIs shall be used. Otherwise a DET-Parameter error shall be emitted.

**NVM436:** The NvM module's environment shall pass a pointer unequal to NULL via the parameter NvM_DstPtr to the function `NvM_RestoreBlockDefaults` in order to copy ROM data to a temporary RAM block although the NVRAM block is configured with a permanent RAM block or explicit synchronization callbacks.

**NVM227:** The job of the function `NvM_RestoreBlockDefaults` shall invalidate a RAM block before copying default data to the RAM if a permanent RAM block is

requested or before explicit synchronization callback (NvMReadRamBlockFromNvM) is called.

**NVM228:** The job of the function `NvM_RestoreBlockDefaults` shall validate and assume a RAM block to be changed if the requested RAM block is permanent or after explicit synchronization callback (NvMReadRamBlockFromNvM) that is called returns E_OK and the copy process of the default data to RAM was successful .

**NVM229:** The job of the function `NvM_RestoreBlockDefaults` shall request a recalculation of CRC from a RAM block after the copy process/validation if a CRC is configured for this RAM block.

Regarding error detection, the requirement NVM027 is applicable to the function `NvM_RestoreBlockDefaults`.

**NVM413:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_RestoreBlockDefaults`.

Hint:
For the block management type `NVM_BLOCK_DATASET`, the application has to ensure that a valid dataset index is selected (pointing to ROM data).

NVRAM common block configuration parameters [NVM028], block management types [NVM061] and one configured NVRAM block descriptor are needed in the configuration with respect to the function `NvM_RestoreBlockDefaults` [NVM062].

#### 8.3.2.4 NvM_EraseNvBlock

**NVM457:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_EraseNvBlock | |
| ***Syntax:*** | `Std_ReturnType NvM_EraseNvBlock( NvM_BlockIdType BlockId )` | |
| ***Service ID[hex]:*** | 0x09 | |
| ***Sync/Async:*** | Asynchronous | |
| ***Reentrancy:*** | Reentrant | |
| ***Parameters (in):*** | BlockId | The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block. |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | None | |
| ***Return value:*** | Std_ReturnType | E_OK: request has been accepted<br>E_NOT_OK: request has not been accepted |
| ***Description:*** | Service to erase a NV block. | |

**NVM415:** The job of the function `NvM_EraseNvBlock` shall erase a NV block.

**NVM231:** The function `NvM_EraseNvBlock` shall take over the given parameters, queue the request and return.

**NVM418:** The function `NvM_EraseNvBlock` shall queue the request to erase in case of disabled write protection.

**NVM416:** The job of the function `NvM_EraseNvBlock` shall leave the content of the RAM block unmodified.

**NVM417:** The function `NvM_EraseNvBlock` shall test the write protection attribute of the NV block in the corresponding Administrative block.

**NVM262:** The function `NvM_EraseNvBlock` shall return with `E_NOT_OK` if a write protected NV block or a ROM block of a dataset NVRAM block is referenced.

**NVM230:** The function `NvM_EraseNvBlock` shall check the write protection of a NVRAM block only before the job is put to the job queue. The NvM module shall not re-check the write protection before fetching the job from the job queue.

**NVM269:** If the referenced NVRAM block is of type `NVM_BLOCK_REDUNDANT`, the function `NvM_EraseNvBlock` shall only succeed when both NV blocks have been erased.

**NVM271:** The job of the function `NvM_EraseNvBlock` shall set the job result to `NVM_REQ_NOT_OK` and report `NVM_E_REQ_FAILED` to the DEM if the processing of the service fails.

**NVM357:** The function `NvM_EraseNvBlock` shall return with `E_NOT_OK`, when development error detection is enabled and the referenced NVRAM block is configured with standard priority.

Regarding error detection, the requirement NVM027 is applicable to the function NvM_EraseNvBlock.

**NVM414:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_EraseNvBlock`.

NVRAM common block configuration parameters [NVM028], block management types [NVM061] and one configured NVRAM block descriptor are needed in the configuration with respect to the function `NvM_EraseNvBlock` [NVM062].

#### 8.3.2.5 NvM_CancelWriteAll

**NVM458:**

| | |
|---|---|
| ***Service name:*** | NvM_CancelWriteAll |
| ***Syntax:*** | `void NvM_CancelWriteAll(`<br><br>`)` |
| ***Service ID[hex]:*** | 0x0a |
| ***Sync/Async:*** | Asynchronous |
| ***Reentrancy:*** | Non Reentrant |
| ***Parameters (in):*** | None |
| ***Parameters (inout):*** | None |
| ***Parameters (out):*** | None |
| ***Return value:*** | None |
| ***Description:*** | Service to cancel a running NvM_WriteAll request. |

**NVM019:** The function `NvM_CancelWriteAll` shall cancel a running `NvM_WriteAll` request. It shall terminate the `NvM_WriteAll` request in a way that the data consistency during processing of a single NVRAM block is not compromised

**NVM232:** The function `NvM_CancelWriteAll` shall signal the request to the NvM module and return.

**NVM233:** The function `NvM_CancelWriteAll` shall be without any effect if no `NvM_WriteAll` request is pending.

**NVM234:** The function `NvM_CancelWriteAll` shall treat multiple requests to cancel a running `NvM_WriteAll` request as one request, i.e. subsequent requests will be ignored.

**NVM235**: The request result of the function `NvM_CancelWriteAll` shall be implicitly given by the result of the `NvM_WriteAll` request to be cancelled.

**NVM255:** The function `NvM_CancelWriteAll` shall ignore an already pending `NvM_CancelWriteAll` request.

**NVM236:** The function `NvM_CancelWriteAll` shall not modify any management information.

Regarding error detection, the requirement NVM027 is applicable to the function NvM_CancelWriteAll.

**NVM419:** The NvM module's environment shall have initialized the NvM module before it calls the function function `NvM_CancelWriteAll`.

**NVM420:** The function `NvM_CancelWriteAll` shall signal the NvM module and shall not be queued, i.e. there can be only one pending request of this type.

#### 8.3.2.6 NvM_InvalidateNvBlock

**NVM459:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_InvalidateNvBlock | |
| ***Syntax:*** | `Std_ReturnType NvM_InvalidateNvBlock( NvM_BlockIdType BlockId )` | |
| ***Service ID[hex]:*** | 0x0b | |
| ***Sync/Async:*** | Asynchronous | |
| ***Reentrancy:*** | Reentrant | |
| ***Parameters (in):*** | BlockId | The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block. |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | None | |
| ***Return value:*** | Std_ReturnType | E_OK: request has been accepted<br>E_NOT_OK: request has not been accepted |
| ***Description:*** | Service to invalidate a NV block. | |

**NVM421:** The job of the function `NvM_InvalidateNvBlock` shall invalidate a NV block.

**NVM422:** The job of the function `NvM_InvalidateNvBlock` shall leave the RAM block unmodified.

**NVM423:** The function `NvM_InvalidateNvBlock` shall check the write protection attribute of the NV block in the administrative part of the corresponding RAM block.

**NVM424:** The function `NvM_InvalidateNvBlock` shall queue the request if the write protection of the corresponding NV block is disabled.

**NVM239:** The function `NvM_InvalidateNvBlock` shall take over the given parameters, queue the request and return.

**NVM272:** The function `NvM_InvalidateNvBlock` shall return with `E_NOT_OK` if a write protected NV block or a ROM block of a dataset NVRAM block is referenced by the BlockId parameter.

**NVM273:** The function `NvM_InvalidateNvBlock` shall check the write protection of a NVRAM block before the job is put into the job queue. The NvM module shall not re-check write protection before fetching the job from the job queue.

**NVM274:** If the referenced NVRAM block is of type `NVM_BLOCK_REDUNDANT`, the function NvM_InvalidateNvBlock shall only set the request result NvM_RequestResultType to NVM_REQ_OK when both NV blocks have been invalidated.

**NVM275:** The function `NvM_InvalidateNvBlock` shall set the job result to `NVM_REQ_NOT_OK` and report `NVM_E_REQ_FAILED` to the DEM if the processing of this service fails.

Regarding error detection, the requirement NVM027 is applicable to the function NvM_InvalidateNvBlock.

**NVM425:** The NvM module's environment shall have initialized the NvM module before it calls the function function `NvM_InvalidateNvBlock`.

NVRAM common block configuration parameters [NVM028], block management types [NVM061] and one configured NVRAM block descriptor are needed in the configuration with respect to the function `NvM_InvalidateBlock` [NVM062].

### 8.3.3 Asynchronous multi block requests

#### 8.3.3.1 NvM_ReadAll

**NVM460:**

| | |
|---|---|
| ***Service name:*** | NvM_ReadAll |
| ***Syntax:*** | `void NvM_ReadAll( )` |
| ***Service ID[hex]:*** | 0x0c |
| ***Sync/Async:*** | Asynchronous |
| ***Reentrancy:*** | Non Reentrant |
| ***Parameters (in):*** | None |
| ***Parameters (inout):*** | None |
| ***Parameters (out):*** | None |
| ***Return value:*** | None |
| ***Description:*** | Initiates a multi block read request. |

**NVM356:** The multi block service `NvM_ReadAll` shall provide two distinct functionalities.

1. Initialize the management data for **all** NVRAM blocks (see NVM304 ff)
2. Copy data to the permanent RAM blocks or call explicit synchronization callback(NvM_ReadRamBlockFromNvm) for those NVRAM blocks which are configured accordingly (see NVM008 ff).

Note: The two functionalities can be implemented in one loop.

**NVM243:** The function `NvM_ReadAll` shall signal the request to the NvM module and return. The NVRAM Manager shall defer the processing of the requested ReadAll until all single block job queues are empty.

**NVM304:** The job of the function `NvM_ReadAll` shall set each proceeding block specific job result for NVRAM blocks and the multi block job result to `NVM_REQ_PENDING` in advance.

**NVM244:** The job of the function `NvM_ReadAll` shall iterate over all user NVRAM blocks, i.e. except for reserved Block Ids 0 (multi block request result) and 1 (NV configuration ID), beginning with the lowest Block Id.

**NVM362:** The NvM module shall initiate the recalculation of the RAM CRC for every NVRAM block with a valid permanent RAM block or explicit synchronization callback configured and `NvmCalcRamBlockCrc == TRUE` during the processing of `NvM_ReadAll`.

**NVM364:** The job of the function `NvM_ReadAll` shall treat the data for every recalculated RAM CRC which matches the stored RAM CRC as valid and set the block specific request result to `NVM_REQ_OK`.

Note: This mechanism enables the NVRAM Manager to avoid overwriting of maybe still valid RAM data with outdated NV data.

**NVM363:** The job of the function `NvM_ReadAll` shall treat invalid marked RAM blocks or the data for every recalculated RAM CRC which doesn't match the stored RAM CRC as invalid and restore the data from NV memory or load default values.

**NVM246:** The job of the function `NvM_ReadAll` shall validate the configuration ID by comparing the stored NVRAM configuration ID vs. the compiled NVRAM configuration ID [NVM034]. The NVRAM block with the block ID 1 (redundant type with CRC) shall be reserved to contain the stored NVRAM configuration ID.

**NVM247:** The job of the function `NvM_ReadAll` shall process the normal runtime preparation for all configured NVRAM blocks and set the error/status information field of the corresponding NVRAM block's administrative block to `NVM_REQ_OK` in case of configuration ID match.

**NVM305:** The job of the function `NvM_ReadAll` shall report the production error `NVM_E_REQ_FAILED` to the DEM and set the error status field of the reserved NVRAM block to `NVM_REQ_INTEGRITY_FAILED` if the configuration ID cannot be read because of an error detected by one of the subsequent SW layers. The NvM module shall behave in the same way as if a configuration ID mismatch was detected.

**NVM307:** The job of the function `NvM_ReadAll` shall set the error/status information field of a NVRAM block's administrative block to `NVM_REQ_NOT_OK` in the case of configuration ID mismatch.

**NVM306:** In case the NvM module can not read the configuration ID because the corresponding NV blocks are empty or invalidated, the job of the function `NvM_ReadAll` shall not report a production error to the DEM but set the error/status information field in this NVRAM block's administrative block to `NVM_REQ_NV_INVALIDATED` and update the configuration Id according to NVM310. The NvM module shall behave the same way as if the configuration ID matched.

**NVM248:** The job of the function `NvM_ReadAll` shall ignore a configuration ID mismatch and behave normal if `NvmDynamicConfiguration == FALSE` [NVM028].

**NVM249:** The job of the function `NvM_ReadAll` shall process the normal runtime preparation of all NVRAM blocks when they are configured with `NvmResistantToChangedSw == TRUE` [NVM061] and

NvmDynamicConfiguration == TRUE [NVM028] and if a configuration ID mismatch occurs. The job of the function NvM_ReadAll shall process an extended runtime preparation for all blocks which are configured with NvmResistantToChangedSw == FALSE.

**NVM314:** The job of the function NvM_ReadAll shall mark every NVRAM block that has not been detected by underlying SW as invalid and is configured with NvmWriteBlockOnce == TRUE as write protected.

Note: The function NvM_SetBlockProtection shall not be able to clear this write protection.

**NVM315:** The job of the function NvM_ReadAll shall only invalidate a NVRAM block of management type NVM_BLOCK_REDUNDANT if both NV blocks have been invalidated.

**NVM008:** The NvM module's environment shall use the multi block request NvM_ReadAll to load and validate the content of configured permanent RAM blocks during start-up [NVM091].

**NVM118:** The job of the function NvM_ReadAll shall process only the permanent RAM blocks or call explicit synchronization callback (NvM_ReadRamBlockFromNvm) for blocks which are configured with NvmSelectBlockForReadall == TRUE.

**NVM287:** The job of the function NvM_ReadAll shall set the job result to NVM_REQ_BLOCK_SKIPPED for all NVRAM blocks which are not loaded automatically during processing of the NvM_ReadAll job.

**NVM426:** If configured by NvmDrvModeSwitch, the job of the function NvM_ReadAll shall switch the mode of each memory device to "fast-mode" before starting to iterate over all user NVRAM bocks.

**NVM427:** If configured by NvmDrvModeSwitch, the job of the function NvM_ReadAll shall switch the mode of each memory device to "slow-mode" after having processed all user NVRAM blocks.

**NVM308:** The job of the function NvM_ReadAll shall load the ROM default data to the corresponding RAM blocks and set the error/status field in the administrative block to NVM_REQ_OK when processing the extended runtime preparation.

**NVM309:** When executing the extended runtime preparation, the job of the function NvM_ReadAll shall treat the affected NVRAM blocks as invalid or blank in order to allow rewriting of blocks configured with NVM_BLOCK_WRITE_ONCE == TRUE.

**NVM310:** The job of the function NvM_ReadAll shall update the configuration ID from the RAM block assigned to the reserved NVRAM block with ID 1 according to

the new (compiled) configuration ID, mark the NVRAM block to be written during NvM_WriteAll and request a CRC recalculation if a configuration ID mismatch occurs and if the NVRAM block is configured with NvmDynamicConfiguration == TRUE.

**NVM311:** The NvM module shall allow applications to send any request for the reserved NVRAM Block, including NvM_WriteBlock, with respect to specified constraints and caveats.

**NVM312:** The NvM module shall not send a request for invalidation of the reserved configuration ID NVRAM block to the underlying layer, unless requested so by the application. This shall ensure that the NvM module's environment can rely on this block to be only invalidated at the first start-up of the ECU or if desired by the application.

**NVM313:** In case of a Configuration ID match, the job of the function NvM_ReadAll shall not automatically write to the Configuration ID block stored in the reserved NVRAM block 1.

**NVM288:** The job of the function NvM_ReadAll shall initiate a read attempt on the second NV block for each NVRAM block of type NVM_BLOCK_REDUNDANT [NVM118] and shall report no error to the DEM if the read attempt on the first NV block fails.

**NVM290:** The job of the function NvM_ReadAll shall set the block specific job result to NVM_REQ_OK if the job has successfully copied the corresponding NV block from NV memory to RAM.

**NVM342:** The job of the function NvM_ReadAll shall set the block specific job result to NVM_REQ_NV_INVALIDATED and shall report no error to the DEM if the MemIf reports MEMIF_BLOCK_INVALID.

**NVM360:** The job of the function NvM_ReadAll shall set the block specific job result to NVM_REQ_INTEGRITY_FAILED and report NVM_E_INTEGRITY_FAILED to the DEM if the MemIf reports MEMIF_BLOCK_INCONSISTENT.

**NVM361:** The job of the function NvM_ReadAll shall set the block specific job result to NVM_REQ_NOT_OK and report NVM_E_REQ_FAILED to the DEM, if the MemIf reports MEMIF_JOB_FAILED.

**NVM291:** The job of the function NvM_ReadAll shall set the block specific job result to NVM_REQ_OK if the corresponding block management type is NVM_BLOCK_REDUNDANT and the function has successfully copied one of the NV blocks from NV memory to RAM.

**NVM292:** The job of the function `NvM_ReadAll` shall request a CRC recalculation over the RAM block data after the copy process [NVM180] if the NV block is configured with CRC.

**NVM293:** The job of the function `NvM_ReadAll` shall load the default values to the RAM blocks according to the processing of `NvM_RestoreBlockDefaults` if the recalculated CRC is not equal to the CRC stored in NV memory or the read request passed to the underlying layer fails. If there are no default values available, the job shall leave the RAM blocks invalid.

**NVM294:** The job of the function `NvM_ReadAll` shall report `NVM_E_INTEGRITY_FAILED` to the DEM if a CRC mismatch occurs.

**NVM295:** The job of the function `NvM_ReadAll` shall set a block specific job result to `NVM_REQ_INTEGRITY_FAILED` if a CRC mismatch occurs.

**NVM302:** The job of the function `NvM_ReadAll` shall report `NVM_E_REQ_FAILED` to the DEM if the referenced NVRAM Block is not configured with CRC and the corresponding job process has failed.

**NVM301:** The job of the function `NvM_ReadAll` shall set the multi block job result to `NVM_REQ_NOT_OK` if the job fails with the processing of at least one NVRAM block.

**NVM281:** If configured by `NvmSingleBlockCallback`, the job of the function `NvM_ReadAll` shall call the single block callback after having completely processed a NVRAM block.

Note: The idea behind using the single block callbacks also for multi-block requests is to speed up the software initialization process:

- A single-block callback issued from a multi-block request (e.g. NvM_ReadAll) will result in an RTE event.
- If the RTE is initialized after or during the asynchronous multi-block request (e.g. NvM_ReadAll), all or some of these RTE events will get lost because they are overwritten during the RTE initialization (see rte_sws_2536).
- After its initialization, the RTE can use the "surviving" RTE events to start software components even before the complete multi-block request (e.g. NvM_ReadAll) has been finished.
- For those RTE events that got lost during the initialization: the RTE will start those software components and the software components either query the status of the NV block they want to access or request that NV block to be read. This is exactly the same behavior if the single-block callbacks would not be used in multi-block requests.

**NVM251:** The job of the function `NvM_ReadAll` shall mark a NVRAM block as “valid/unmodified” if NV data has been successfully loaded to the RAM Block.

**NVM367:** The job of the function `NvM_ReadAll` shall set a RAM block to valid and assume it to be changed if the job has successfully copied default values to the corresponding RAM.

**NVM428:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_ReadAll`.

**NVM429:** In Development Mode, the NvM module shall reject the function `NvM_ReadAll` if another multi block request is pending.

Regarding error detection, the requirement NVM027 is applicable to the function `NvM_ReadAll`.

The DEM shall already be able to accept error notifications.

NVRAM common block configuration parameters [NVM028], block management types [NVM061] and all configured NVRAM block descriptors are needed in the configuration with respect to the function `NvM_ReadAll` [NVM062], [NVM069].

#### 8.3.3.2 NvM_WriteAll

**NVM461:**

| | |
|---|---|
| ***Service name:*** | NvM_WriteAll |
| ***Syntax:*** | `void NvM_WriteAll(`<br><br>`)` |
| ***Service ID[hex]:*** | 0x0d |
| ***Sync/Async:*** | Asynchronous |
| ***Reentrancy:*** | Non Reentrant |
| ***Parameters (in):*** | None |
| ***Parameters (inout):*** | None |
| ***Parameters (out):*** | None |
| ***Return value:*** | None |
| ***Description:*** | Initiates a multi block write request. |

**NVM018:** The job of the function `NvM_WriteAll` shall synchronize the contents of permanent RAM blocks to their corresponding NV blocks or call explicit synchronization callback (NvM_WriteRamBlockToNvm) on shutdown.

**NVM733:** If NVRAM block ID 1 (which holds the configuration ID of the memory layout) is marked as "to be written during NvM_WriteAll", the job of the function NvM_WriteAll shall write this block in a final step (last write operation) to prevent memory layout mismatch in case of a power loss failure during write operation.

**NVM254:** The function `NvM_WriteAll` shall signal the request to the NvM module and return. The NVRAM Manager shall defer the processing of the requested WriteAll until all single block job queues are empty.

**NVM252:** The job of the function `NvM_WriteAll` shall process all permanent RAM blocks or call explicit synchronization callback (NvM_WriteRamBlockToNvm) except for block ID 0.

**NVM430:** If configured by `NvmDrvModeSwitch`, the job of the function `NvM_WriteAll` shall set the mode of each memory device to “fast-mode” before starting to iterate over all non-reserved NVRAM blocks.

**NVM431:** If configured by `NvmDrvModeSwitch`, the job of the function `NvM_WriteAll` shall set the mode of each memory device to “slow-mode” after having processed all non-reserved NVRAM blocks or after the function `NvM_CancelWriteAll` has cancelled the job.

**NVM432:** The job of the function `NvM_WriteAll` shall check the write-protection and “valid/modified” state for each RAM block in advance.

**NVM433:** The job of the function NvM_WriteAll shall only write the content of a RAM block to its corresponding NV block for non write-protected NVRAM blocks.

**NVM474:** The job of the function NvM_WriteAll may write unchanged data, if this would repair (redundant) NV data. Otherwise it should not write unchanged data.

**NVM747:** The job of the function NvM_WriteAll shall copy the data content of the RAM block to both corresponding NV blocks if the NVRAM block management type of the processed NVRAM block is NVM_BLOCK_REDUNDANT.

**NVM748:** If the processed NVRAM block is of type NVM_BLOCK_REDUNDANT the job of the function NvM_WriteAll shall start to copy the data of the RAM block to NV block which has not been read during the job started by NvM_ReadBlock or NvM_ReadAll then continue to copy the other NV block.

**NVM434:** The job of the function NvM_WriteAll shall skip every write-protected NVRAM block without error notification.

**NVM298:** The job of the function NvM_WriteAll shall set the job result for each NVRAM block which has not been written automatically by the job to NVM_REQ_BLOCK_SKIPPED.

**NVM339:** In case of NVRAM block management type NVM_BLOCK_DATASET, the job of the function NvM_WriteAll shall copy only the RAM block to the corresponding NV block which is selected via the data index in the administrative block.

**NVM253:** The job of the function NvM_WriteAll shall request a CRC recalculation and renew the CRC from a NVRAM block before writing the data if a CRC is configured for this NVRAM block.

**NVM296:** The job of the function NvM_WriteAll shall check the number of write retries [NVM028] by a write retry counter to avoid infinite loops. Each unsuccessful result reported by the MemIf module shall be followed by an increment of the retry counter. The job of the function NvM_WriteAll shall set the block specific job result to NVM_REQ_NOT_OK and report NVM_E_REQ_FAILED to the DEM if the write retry counter becomes greater than the configured NVM_MAX_NO_OF_WRITE_RETRIES.

**NVM337:** The job of the function NvM_WriteAll shall set the single block job result to NVM_REQ_OK if the processed NVRAM block is of type NVM_BLOCK_REDUNDANT and at least one of the NV blocks has been written successfully.

**NVM238:** The job of the function NvM_WriteAll shall complete the job in a non-destructive way for the NVRAM block currently being processed if a cancellation of NvM_WriteAll is signaled by a call of NvM_CancelWriteAll.

**NVM237:** The NvM module shall set the multi block request result to `NVM_REQ_CANCELLED` in case of cancellation of `NvM_WriteAll`. The NvM module shall anyway report the error code condition, due to a failed NVRAM block write, to the DEM.

**NVM318:** The job of the function `NvM_WriteAll` shall set the multi block request result to `NVM_REQ_NOT_OK` if processing of one or even more NVRAM blocks fails.

**NVM329:** If the job of the function `NvM_WriteAll` has successfully written data to NV memory for a NVRAM block configured with `NvmWriteBlockOnce == TRUE`, the job shall immediately set the corresponding write protection flag in the administrative block.

Regarding error detection, the requirement NVM027 is applicable to the function NvM_WriteAll.

**NVM473:** In case of a RAM block is successfully copied to NV memory the RAM block state shall be set to "valid/unmodified" afterwards.

**NVM437:** The NvM module's environment shall have initialized the NvM module before it calls the function `NvM_WriteAll`.

No other multiblock request shall be pending when the NvM module's environment calls the function `NvM_WriteAll`.

Note: To avoid the situation that in case of redundant NVRAM blocks two different NV blocks are containing different but valid data at the same time, each client of the `NvM_WriteAll` service may call `NvM_InvalidateNvBlock` in advance.

NVRAM common block configuration parameters [NVM028], block management types [NVM061] and all configured NVRAM block descriptors are needed in the configuration with respect to the `NvM_WriteAll` function [NVM062], [NVM069].

## 8.4 Call-back notifications

### 8.4.1 Callback notification of the NvM module

**NVM438:** The NvM module shall provide callback functions to be used by the underlying memory abstraction (EEPROM abstraction / FLASH EEPROM Emulation) to signal end of job state with or without error.

**NVM439:** The file `NvM_Cbk.h` shall provide the function prototypes of the callback functions.

Note: The file `NvM_Cbk.h` is to be included by the underlying memory driver layers.

#### 8.4.1.1 NVRAM Manager job end notification without error

**NVM462:**

| | |
|---|---|
| ***Service name:*** | NvM_JobEndNotification |
| ***Syntax:*** | `void NvM_JobEndNotification(`<br><br>`)` |
| ***Service ID[hex]:*** | 0x00 |
| ***Sync/Async:*** | Synchronous |
| ***Reentrancy:*** | Non Reentrant |
| ***Parameters (in):*** | None |
| ***Parameters (inout):*** | None |
| ***Parameters (out):*** | None |
| ***Return value:*** | None |
| ***Description:*** | Function to be used by the underlying memory abstraction to signal end of job without error. |

**NVM111:** The callback function `NvM_JobEndNotification` is used by the underlying memory abstraction to signal end of job without error.

Note: Successful job end notification of the memory abstraction:

- Read finished & OK
- Write finished & OK
- Erase finished & OK

This routine might be called in interrupt context, depending on the calling function. All memory abstraction modules should be configured to use the same mode (callback/polling).

**NVM440:** The NvM module shall only provide the callback function `NvM_JobEndNotification` if polling mode is disabled via `NvmPollingMode`.

The function `NvM_JobEndNotification` is affected by the common [NVM028] configuration parameters.

#### 8.4.1.2 NVRAM Manager job end notification with error

**NVM463:**

| | |
|---|---|
| ***Service name:*** | NvM_JobErrorNotification |
| ***Syntax:*** | `void NvM_JobErrorNotification(`<br><br>`)` |
| ***Service ID[hex]:*** | 0x00 |
| ***Sync/Async:*** | Synchronous |

| | |
|---|---|
| ***Reentrancy:*** | Non Reentrant |
| ***Parameters (in):*** | None |
| ***Parameters (inout):*** | None |
| ***Parameters (out):*** | None |
| ***Return value:*** | None |
| ***Description:*** | Function to be used by the underlying memory abstraction to signal end of job with error. |

**NVM112:** The callback function `NvM_JobErrorNotification` is to be used by the underlying memory abstraction to signal end of job with error.

Note: Unsuccessful job end notification of the memory abstraction:

- Read aborted or failed
- Write aborted or failed
- Erase aborted or failed

This routine might be called in interrupt context, depending on the calling function. All memory abstraction modules should be configured to use the same mode (callback/polling).

**NVM441:** The NvM module shall only provide the callback function `NvM_JobErrorNotification` if polling mode is disabled via `NvmPollingMode`.

The function `NvM_JoberrorNotification` is affected by the common [NVM028] configuration parameters.

## 8.5 Scheduled functions

These functions are directly called by the Basic Software Scheduler. The following functions shall have no return value and no parameter. All functions shall be non reentrant.

.**NVM322:** BSW module main processing functions are only allowed to be allocated to basic tasks by the function NvM_MainFunction.

**NVM464:**

| | |
|---|---|
| ***Service name:*** | NvM_MainFunction |
| ***Syntax:*** | `void NvM_MainFunction(`<br><br>`)` |
| ***Service ID[hex]:*** | 0x0e |
| ***Timing:*** | VARIABLE_CYCLIC |
| ***Description:*** | Service for performing the processing of the NvM jobs. |

**NVM256:** The function `NvM_MainFunction` shall perform the processing of the NvM module jobs.

**NVM333:** The function `NvM_MainFunction` shall perform the CRC recalculation if requested for a NVRAM block in addition to NVM256.

**NVM334:** The NvM module shall only start writing of a block (i.e. hand over the job to the lower layers) after CRC calculation for this block has been finished.

**NVM257:** The NvM module shall only do/start job processing, queue management and CRC recalculation if the `NvM_Init` function has internally set an “INIT DONE” signal.

**NVM258:** The function `NvM_MainFunction` shall restart a destructively cancelled request caused by an immediate priority request after the NvM module has processed the immediate priority request [NVM152].

**NVM259:** The function `NvM_MainFunction` shall supervise the immediate priority queue (if configured) regarding the existence of immediate priority requests.

**NVM346:** If polling mode is enabled, the function `NvM_MainFunction` shall check the status of the requested job sent to the lower layer.

**NVM347:** If callback routines are configured, the function `NvM_MainFunction` shall call callback routines to the upper layer after completion of an asynchronous service.

**NVM350:** In case of processing an `NvM_WriteAll` multi block request, the function `NvM_MainFunction` shall not call callback routines to the upper layer as long as the service `MemIf_GetStatus` returns `MEMIF_BUSY_INTERNAL` for the reserved device ID `MEMIF_BROADCAST_ID` [6]. For this purpose (status is `MEMIF_BUSY_INTERNAL`), the function `NvM_MainFunction` shall cyclically poll the status of the Memory Hardware Abstraction independent of being configured for polling or callback mode.

**NVM349:** The function `NvM_MainFunction` shall return immediately if no further job processing is possible.

Note: NVRAM blocks with immediate priority are not expected to be configured to have a CRC.

**NVM324:** The NvM module’s environment does not have to execute the function `NvM_MainFunction` in a specific order or sequence with respect to other BSW main processing function(s).

The function NvM_MainFunction is affected by the common [NVM028] configuration parameters.

Terms and definitions:

**Fixed cyclic**: Fixed cyclic means that one cycle time is defined at configuration and shall not be changed because functionality is requiring that fixed timing (e.g. filters).

**Variable cyclic**: Variable cyclic means that the cycle times are defined at configuration but might be mode dependent and therefore vary during runtime.

**On pre-condition**: On pre-condition means that no cycle time can be defined. The function is called when the conditions are fulfilled. Alternatively, the function may be called cyclically. However, the cycle time is assigned dynamically during runtime by other modules.

## 8.6 Expected Interfaces

In this chapter, all interfaces required by other modules are listed.

### 8.6.1 Mandatory Interfaces

**NVM319:** The following table defines all interfaces which are required to fulfill the core functionality of the module.

**NVM465:**

| *API function* | *Description* |
|---|---|
| Dem_ReportErrorStatus | Reports errors to the DEM. |
| MemIf_Cancel | map function calls of MemIf_Cancel to service: Fee_Cancel respectively Ea_Cancel |
| MemIf_EraseImmediateBlock | map function calls of MemIf_EraseImmediateBlock to service: Fee_EraseImmediateBlock respectively Ea_EraseImmediateBlock |
| MemIf_GetJobResult | map function calls of MemIf_GetJobResult to service: Fee_GetJobResult respectively Ea_GetJobResult |
| MemIf_GetStatus | map function calls of MemIf_GetStatus to service: Fee_GetStatus respectively Ea_GetStatus |
| MemIf_InvalidateBlock | map function calls of MemIf_InvalidateBlock to service: Fee_InvalidateBlock respectively Ea_InvalidateBlock |
| MemIf_Read | map function calls of MemIf_Read to service: Fee_Read respectively Ea_Read |
| MemIf_Write | map function calls of MemIf_Write to service: Fee_Write respectively Ea_Write |

### 8.6.2 Optional Interfaces

**NVM320:** The following table defines all interfaces which are required to fulfill an optional functionality of the module.

**NVM466:**

| *API function* | *Description* |
|---|---|
| Crc_CalculateCRC16 | This service makes a CRC16 calculation on Crc_Length data bytes. |

| | |
|---|---|
| Crc_CalculateCRC32 | This service makes a CRC32 calculation on Crc_Length data bytes. |
| Det_ReportError | Service to report development errors. |
| EcuM_CB_NfyNvMJobEnd | Used to notify about the end of NVRAM jobs initiated by EcuM<br>The callback must be callable from normal and interrupt execution contexts. |
| MemIf_SetMode | map function calls of MemIf_SetMode to service: Fee_SetMode respectively Ea_SetMode |

### 8.6.3 Configurable interfaces

In this chapter, all interfaces are listed for which the target function can be configured. The target function is usually a callback function. The names of these interfaces are not fixed because they are configurable.

**NVM113:** The notification of a caller via an asynchronous callback routine (NvmSingleBlockCallback) shall be optionally configurable for all NV blocks (see NVM061).

**NVM740:** If a callback is configured for a NVRAM block, every asynchronous block request to the block itself shall be terminated with an invocation of the callback routine.

**NVM741:** The ID identifying the NVRAM service, shall be passed to the callback routine.

**NVM742:** If no callback is configured for a NVRAM block, there shall be no asynchronous notification of the caller in case of an asynchronous block request.

**NVM260:** A common callback entry (`NvmMultiBlockCallback`) which is not bound to any NVRAM block [NVM028] shall be optionally configurable for all asynchronous multi block requests (including `NvM_CancelWriteAll`). The ID identifying the NVRAM service shall be passed to the common callback routine. If a NULL pointer is configured for the common callback entry, there shall be no asynchronous notification of the caller in case of asynchronous multi block requests (including `NvM_CancelWriteAll`).

#### 8.6.3.1 Single block job end notification

**NVM467:**

| | | |
|---|---|---|
| ***Service name:*** | SingleBlockCallbackFunction | |
| ***Syntax:*** | `Std_ReturnType SingleBlockCallbackFunction( uint8 ServiceId, NvM_RequestResultType JobResult )` | |
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Non Reentrant | |
| ***Parameters (in):*** | ServiceId | Unique Service ID of NVRAM manager service. |
| | JobResult | Covers the job result of the previous processed single block |

| | | |
|---|---|---|
| | | job. |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | None | |
| ***Return value:*** | Std_ReturnType | E_OK: Callback function has been processed successfully.<br>E_NOT_OK: Callback function has not been processed successfully. |
| ***Description:*** | Per block callback routine to notify the upper layer that an asynchronous single block request has been finished. | |

**NVM368:** The single block callback function shall always return with `E_OK`.

There is no need for the NvM module to evaluate the return value of the single block callback function because of NVM368.

**NVM330:** The single block callback function shall be a function pointer.

Note: Please refer to `NvmSingleBlockCallback` in chapter 10.

The Single block job end notification might be called in interrupt context, depending on the calling function.

### 8.6.3.2 Multi block job end notification

**NVM468:**

| | | |
|---|---|---|
| ***Service name:*** | MultiBlockCallbackFunction | |
| ***Syntax:*** | `void MultiBlockCallbackFunction( uint8 ServiceId, NvM_RequestResultType JobResult )` | |
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Non Reentrant | |
| ***Parameters (in):*** | ServiceId | Unique Service ID of NVRAM manager service. |
| | JobResult | Covers the job result of the previous processed multi block job. |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | None | |
| ***Return value:*** | None | |
| ***Description:*** | Common callback routine to notify the upper layer that an asynchronous multi block request has been finished. | |

**NVM331:** The Multi block job end notification shall be a function pointer.

Note: Please refer to `NvmMultiBlockCallback` in chapter 10.

The Multi block job end notification might be called in interrupt context, depending on the calling function.

### 8.6.3.3 Callback function for block initialization

**NVM469:**

| | | |
|---|---|---|
| ***Service name:*** | InitBlockCallbackFunction | |
| ***Syntax:*** | `Std_ReturnType InitBlockCallbackFunction( )` | |
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Non Reentrant | |
| ***Parameters (in):*** | None | |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | None | |
| ***Return value:*** | Std_ReturnType | E_OK: callback function has been processed successfully<br>E_NOT_OK: callback function has not been processed successfully |
| ***Description:*** | Per block callback routine which shall be called by the NvM module when default data needs to be restored in RAM, even if a ROM block is configured.<br>Note: Here the application should copy default data to a RAM block if a ROM block isn't configured and/or it could set some flags to know that default data was restored. | |

**NVM369:** The Init block callback for block initialization shall always return with `E_OK`.

There is no need for the NvM module to evaluate the return value of the callback function because of NVM369.

**NVM352:** The Init block callback shall be a function pointer.

Note: Please refer to `NvmInitBlockCallback` in chapter 10.

The init block callback function might be called in interrupt context.

#### 8.6.3.4 Callback function for RAM to NvM copy

**NVM539:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_WriteRamBlockToNvm | |
| ***Syntax:*** | `Std_ReturnType NvM_WriteRamBlockToNvm( void* NvMBuffer )` | |
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Non Reentrant | |
| ***Parameters (in):*** | None | |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | NvMBuffer | the address of the buffer where the data shall be written to |
| ***Return value:*** | Std_ReturnType | E_OK: callback function has been processed successfully<br>E_NOT_OK: callback function has not been processed successfully |
| ***Description:*** | Block specific callback routine which shall be called in order to let the application copy data from RAM block to NvM module's mirror. | |

**NVM541:** The RAM to NvM copy callback shall be a function pointer.
Note: Please refer to NvMWriteRamBlockToNvM in chapter 10.

#### 8.6.3.5 Callback function for NvM to RAM copy

**NVM540:**

| | | |
|---|---|---|
| ***Service name:*** | NvM_ReadRamBlockFromNvm | |
| ***Syntax:*** | `Std_ReturnType NvM_ReadRamBlockFromNvm( const void* NvMBuffer )` | |
| ***Sync/Async:*** | Synchronous | |
| ***Reentrancy:*** | Non Reentrant | |
| ***Parameters (in):*** | NvMBuffer | the address of the buffer where the data can be read from |
| ***Parameters (inout):*** | None | |
| ***Parameters (out):*** | None | |
| ***Return value:*** | Std_ReturnType | E_OK: callback function has been processed successfully<br>E_NOT_OK: callback function has not been processed successfully |
| ***Description:*** | Block specific callback routine which shall be called in order to let the application copy data from NvM module's mirror to RAM block. | |

**NVM542:** The NvM to RAM copy callback shall be a function pointer.
Note: Please refer to NvMReadRamBlockFromNvM in chapter 10.

## 8.7 API Overview

| ***Request Types*** | ***Characteristics of Request Types*** |
|---|---|
| Type 1:<br><br>- NvM_SetDataIndex (...)<br>- NvM_GetDataIndex (...)<br>- NvM_SetBlockProtection (...)<br>- NvM_GetErrorStatus(...)<br>- NvM_SetRamBlockStatus(...) | - synchronous request<br>- affects one RAM block<br>- available for all SW-Cs |
| Type 2:<br><br>- NvM_ReadBlock(...)<br>- NvM_WriteBlock(...)<br>- NvM_RestoreBlockDefaults(...)<br>- NvM_EraseNvBlock(...)<br>- NvM_InvalidateNvBlock(...) | - asynchronous request (result via callback or polling)<br>- affects one NVRAM block<br>- handled by NVRAM manager task via request list<br>- available for all SW-Cs |
| Type 3:<br><br>- NvM_ReadAll(...)<br>- NvM_WriteAll(...)<br>- NvM_CancelWriteAll(...) | - asynchronous request (result via callback or polling)<br>- affects all NVRAM blocks with permanent RAM data |
| Type 4:<br><br>- NvM_Init(...) | - synchronous request<br>- basic initialization<br>- success signaled to the task via command interface inside the function itself |

# 9 Sequence Diagrams

## 9.1 Synchronous calls

### 9.1.1 NvM_Init

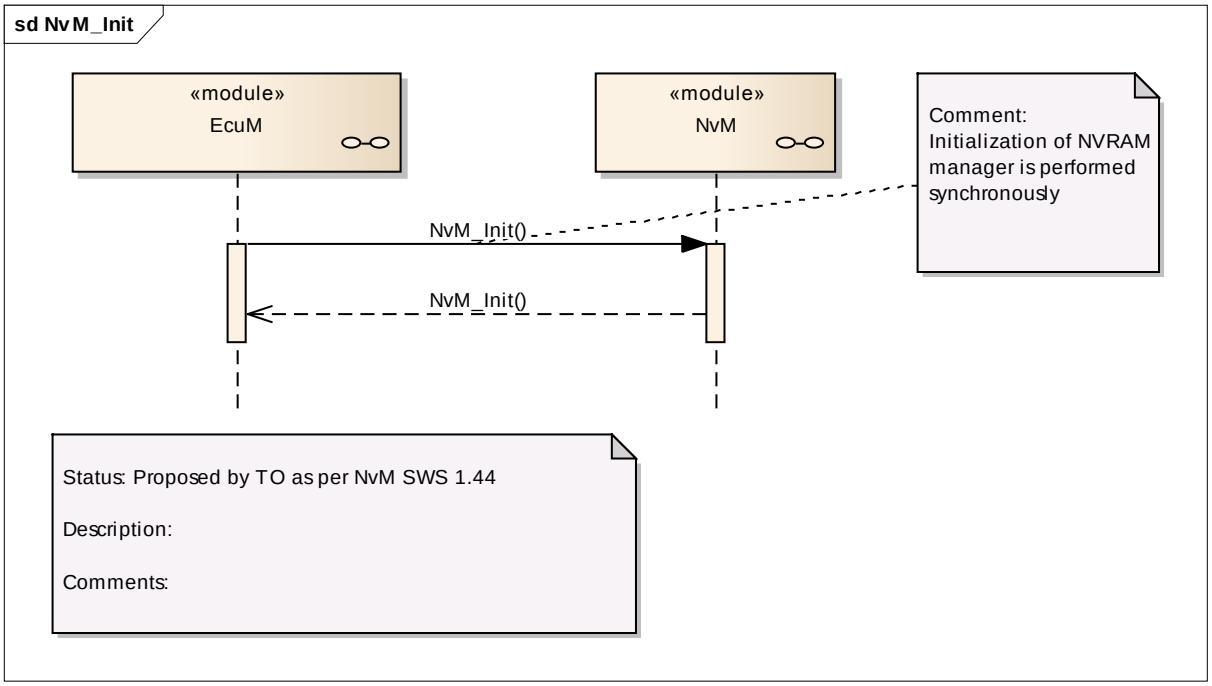


**Figure 10: UML sequence diagram NvM_Init**

### 9.1.2 NvM_SetDataIndex

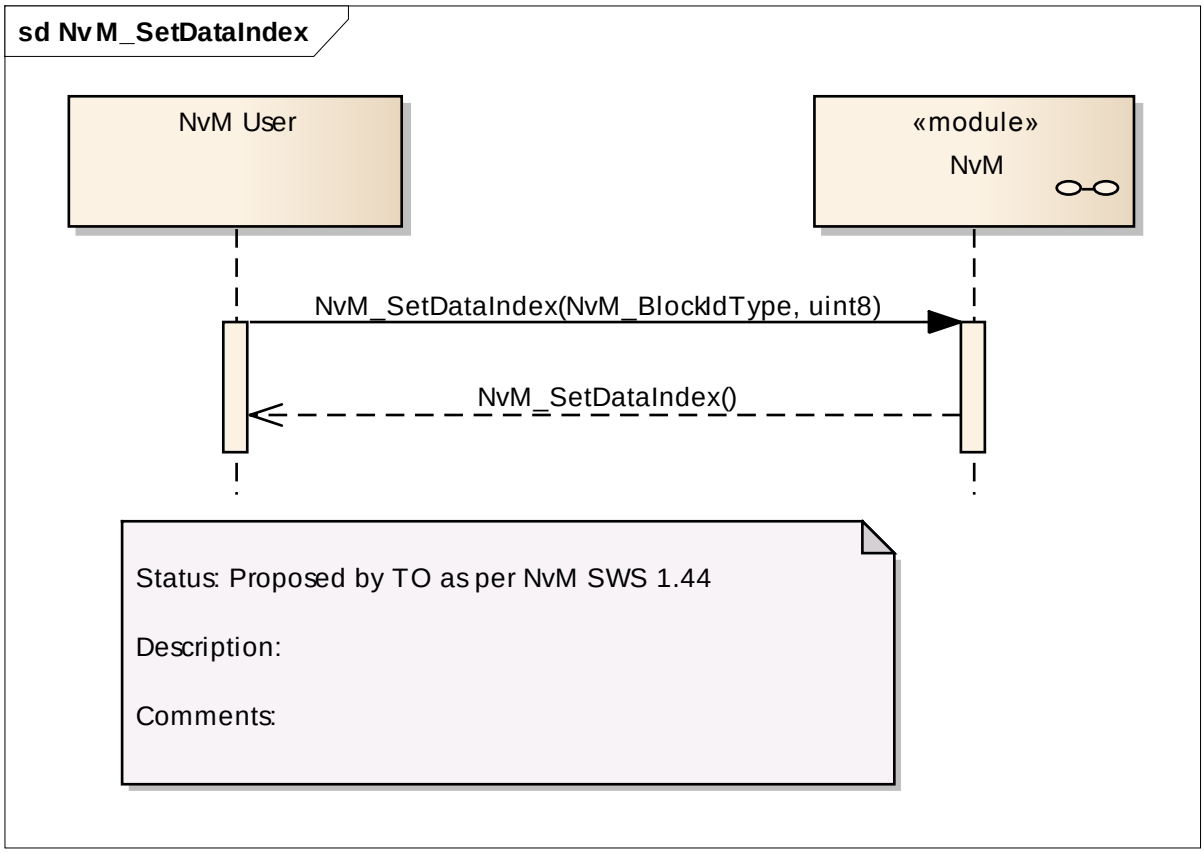


**Figure 11: UML sequence diagram NvM_SetDataIndex**

### 9.1.3 NvM_GetDataIndex

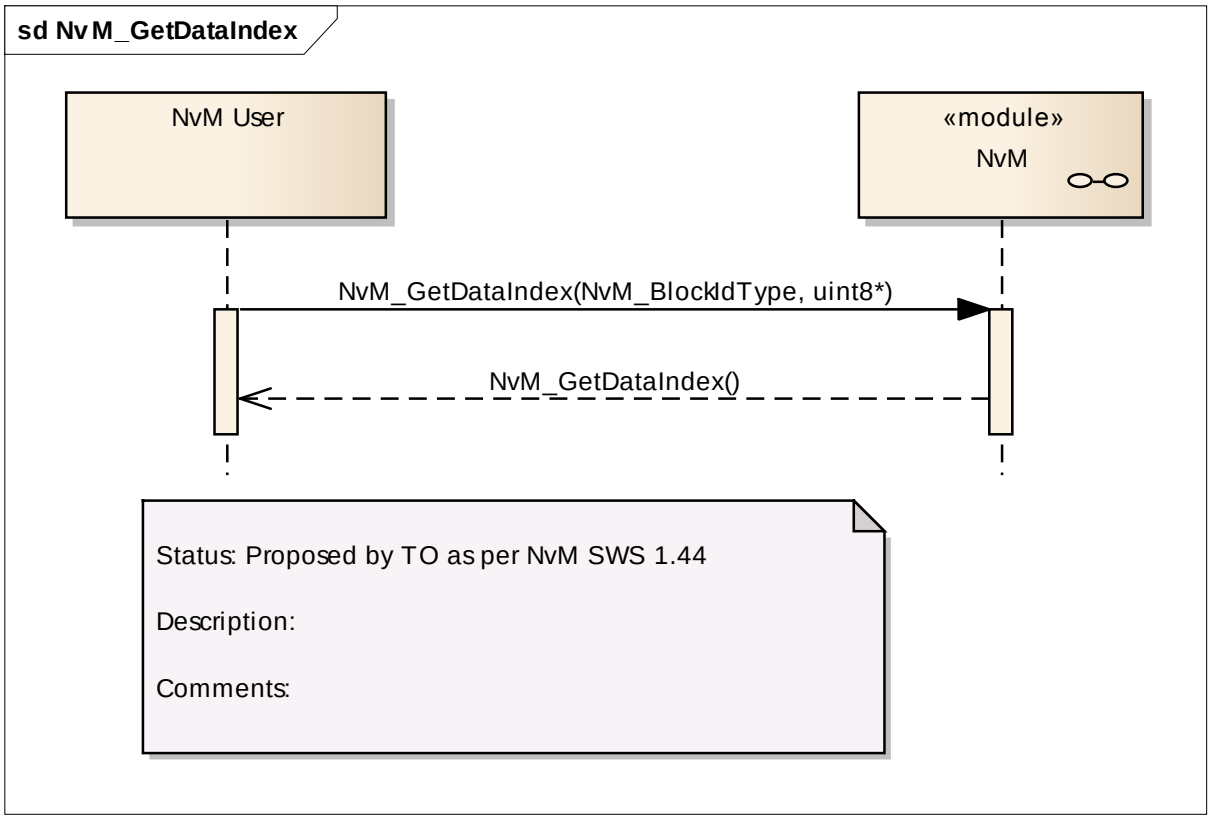


**Figure 12: UML sequence diagram NvM_GetDataIndex**

### 9.1.4 NvM_SetBlockProtection

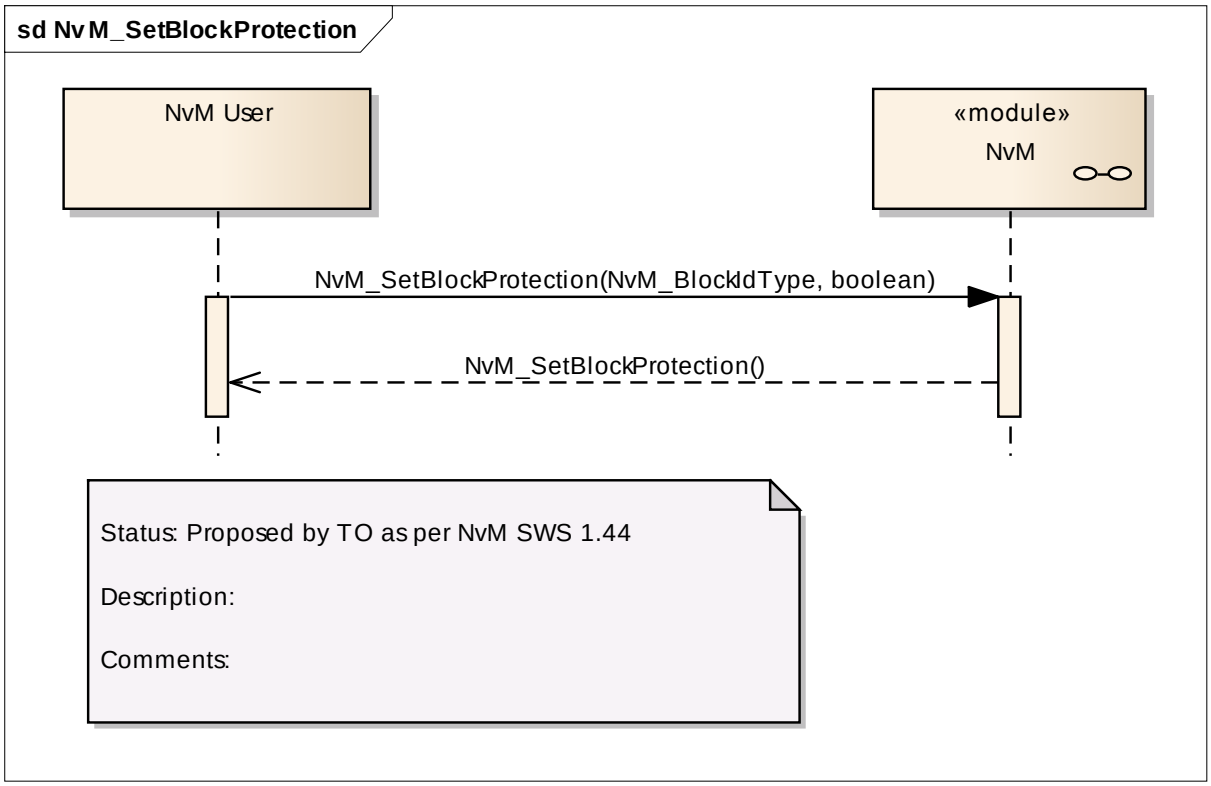


**Figure 13: UML sequence diagram NvM_SetBlockProtection**

### 9.1.5 NvM_GetErrorStatus

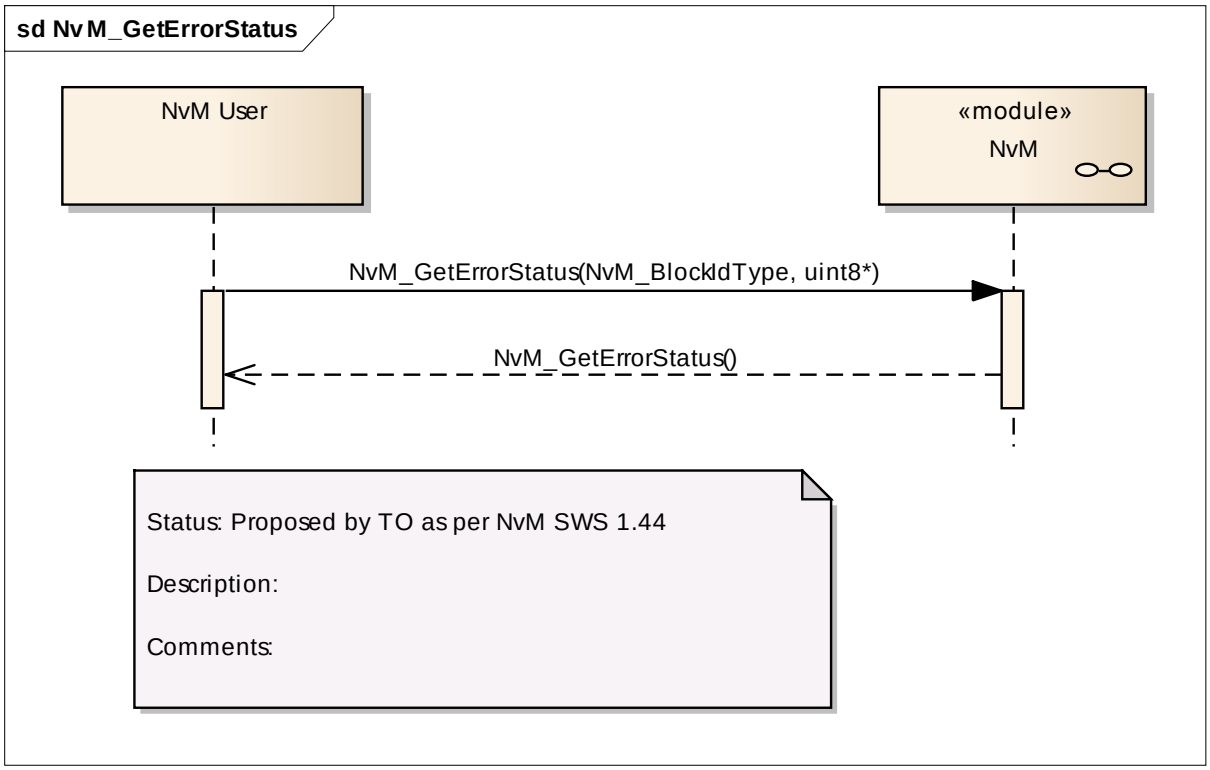


**Figure 14: UML sequence diagram NvM_GetErrorStatus**

### 9.1.6 NvM_GetVersionInfo

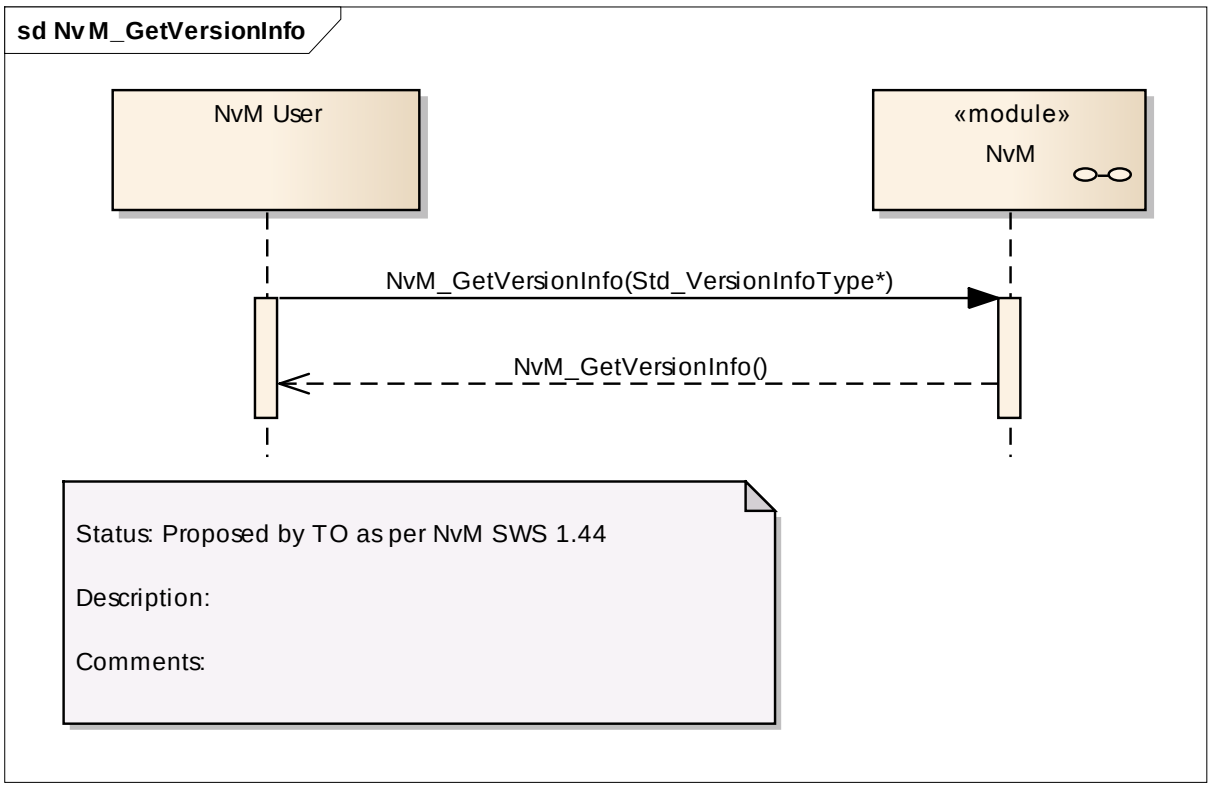

**Figure 15: UML sequence diagram NvM_GetVersionInfo**

## 9.2 Asynchronous calls

The following sequence diagrams concentrate on the interaction between the NvM module and SW-C's or the ECU state manager. For interaction regarding the Memory Interface please ref. to [4] or [5].

### 9.2.1 Asynchronous call with polling

The following diagram shows the function `NvM_WriteBlock` as an example of a request that is performed asynchronously. The sequence for all other asynchronous functions is the same, only the processed number of blocks and the block types may vary. The result of the asynchronous function is obtained by polling requests to the error/status information.

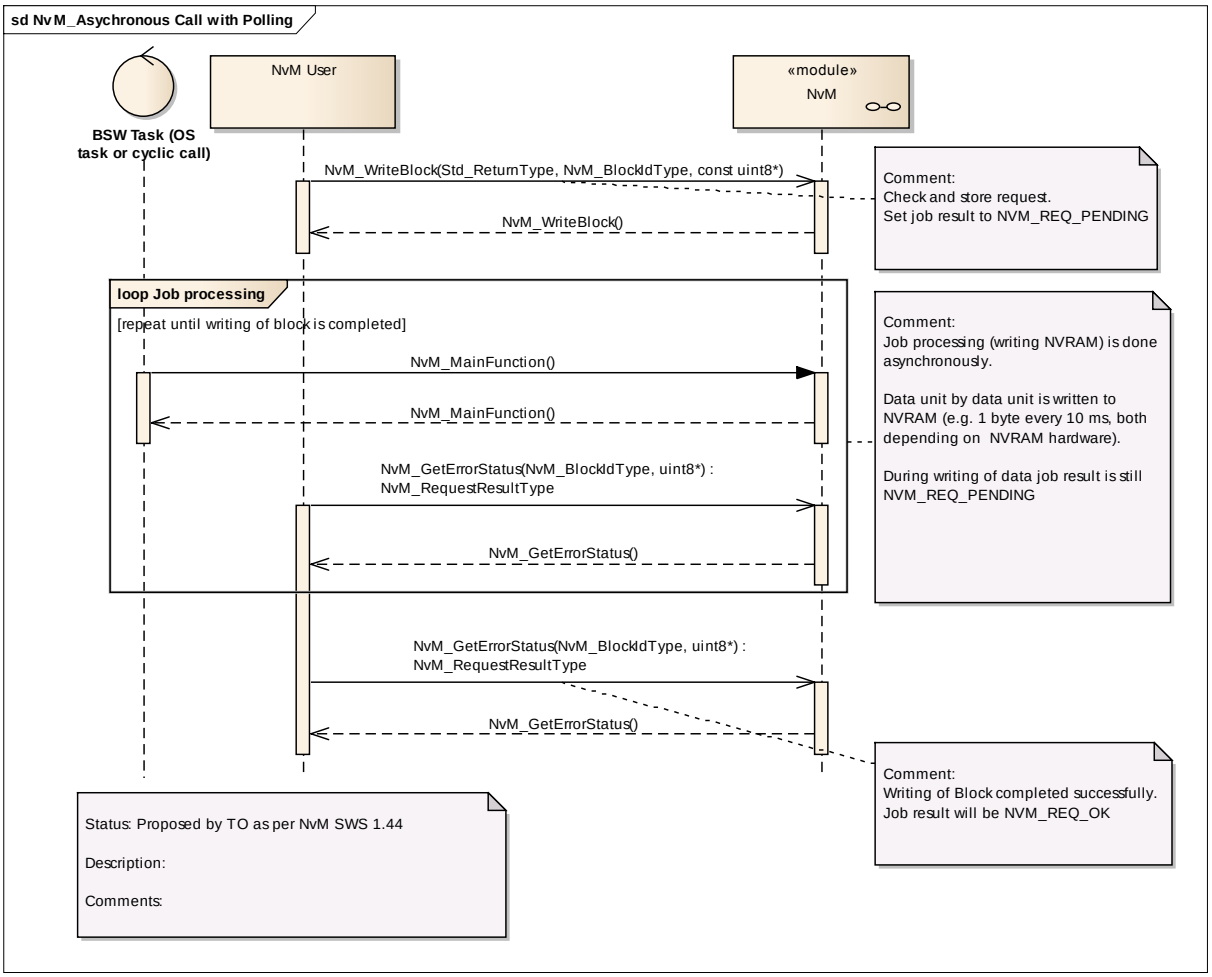


**Figure 16: UML sequence diagram for asynchronous call with polling**

### 9.2.2 Asynchronous call with callback

The following diagram shows the function `NvM_WriteBlock` as an example of a request that is performed asynchronously. The sequence for all other asynchronous functions is the same, only the processed number of blocks and the block types may vary. The result of the asynchronous function is obtained after an asynchronous notification (callback) by requesting the error/status information.

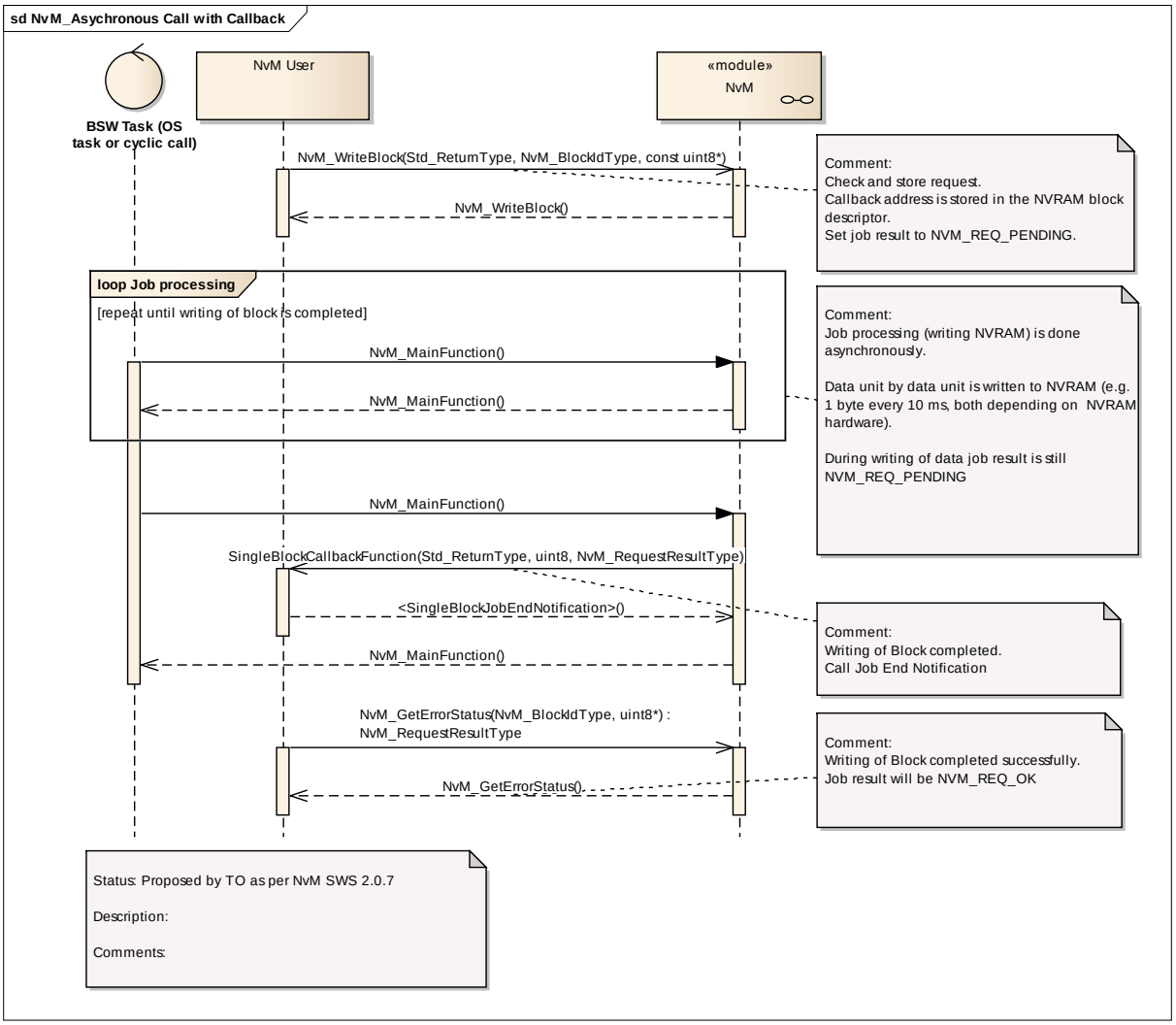


**Figure 17: UML sequence diagram for asynchronous call with callback**

### 9.2.3 Cancellation of a Multi Block Request

The following diagram shows the effect of a cancel operation applied to a running `NvM_WriteAll` multi block request. The running `NvM_WriteAll` function completes the actual NVRAM block and stops further writes.

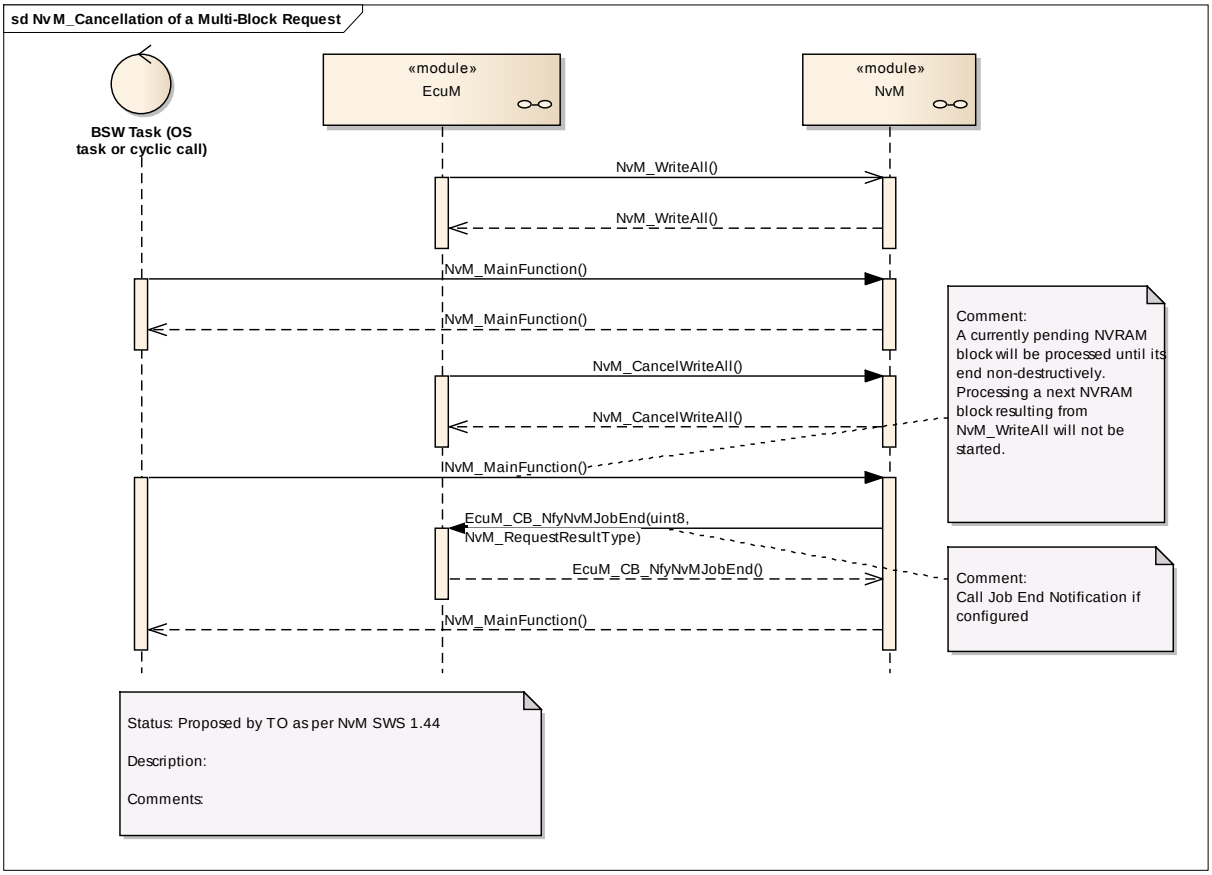


**Figure 18: UML sequence diagram for cancellation of asynchronous call**

# 10 Configuration specification

In general, this chapter defines configuration parameters and their clustering into containers. In order to support the specification Chapter 10.1 describes fundamentals. It also specifies a template (table) you shall use for the parameter specification.

Chapter 10.2 specifies the structure (containers) and the parameters of the module NvM.

Chapter 0 specifies published information of the module NvM.

## 10.1 How to read this chapter

In addition to this section, it is highly recommended to read the documents:

- AUTOSAR Layered Software Architecture [1]
- AUTOSAR ECU Configuration Specification [11]
  This document describes the AUTOSAR configuration methodology and the AUTOSAR configuration metamodel in detail.

The following is only a short survey of the topic and it will not replace the ECU Configuration Specification document.

### 10.1.1 Configuration and configuration parameters

Configuration parameters define the variability of the generic part(s) of an implementation of a module. This means that only generic or configurable module implementation can be adapted to the environment (software/hardware) in use during system and/or ECU configuration.

The configuration of parameters can be achieved at different times during the software process: before compile time, before link time or after build time. In the following, the term “configuration class” (of a parameter) shall be used in order to refer to a specific configuration point in time.

### 10.1.2 Variants

Variants describe sets of configuration parameters. E.g., variant 1: only pre-compile time configuration parameters; variant 2: mix of pre-compile- and post build time-configuration parameters. In one variant a parameter can only be of one configuration class.

### 10.1.3 Containers

Containers structure the set of configuration parameters. This means:

- *all* configuration parameters are kept in containers.

- (sub-) containers can reference (sub-) containers. It is possible to assign a multiplicity to these references. The multiplicity then defines the possible number of instances of the contained parameters.

## 10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters. The detailed meanings of the parameters describe chapter 7.2 and chapter 8.

### 10.2.1 Variants

Variant1: This variant allows only pre-compile time configuration parameters.

### 10.2.2 NvM

| | |
|---|---|
| ***Module Name*** | *NvM* |
| ***Module Description*** | Configuration of the NvM (NvRam Manager) module. |

| ***Included Containers*** | | |
|---|---|---|
| ***Container Name*** | ***Multiplicity*** | ***Scope / Dependency*** |
| NvmBlockDescriptor | 1..65536 | Container for a management structure to configure the composition of a given NVRAM Block Management Type. Its multiplicity describes thenumber of configured NVRAM blocks, one block is required to be configured. The NVRAM block descriptors are condensed in the NVRAM block descriptor table. |
| NvmCommon | 1 | Container for common configuration options. |

### 10.2.3 NvmCommon

| | |
|---|---|
| ***SWS Item*** | **NVM028 :** |
| ***Container Name*** | NvmCommon |
| ***Description*** | Container for common configuration options. |
| ***Configuration Parameters*** | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM555 :** | | |
| ***Name*** | NvMMainFunctionPeriod {NVM_MAIN_FUNCTION_PERIOD} | | |
| ***Description*** | Allows to configure the period for the MainFunction (in seconds). Please note that the range shall be greater than 0. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | FloatParamDef | | |
| ***Range*** | 1E-7 .. INF | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: ECU | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM491 :** | | |
| ***Name*** | NvmApiConfigClass {NVM_API_CONFIG_CLASS} | | |
| ***Description*** | Preprocessor switch to enable some API calls which are related to NVM API configuration classes. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | EnumerationParamDef | | |
| ***Range*** | NVM_API_CONFIG_CLASS_1 | All API calls belonging to configuration class 1 are available. | |
| | NVM_API_CONFIG_CLASS_2 | All API calls belonging to configuration class 2 are available. | |
| | NVM_API_CONFIG_CLASS_3 | All API calls belonging to configuration class 3 are available. | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM492 :** | | |
| ***Name*** | NvmCompiledConfigId {NVM_COMPILED_CONFIG_ID} | | |
| ***Description*** | Configuration ID regarding the NV memory layout. This configuration ID shall be published as e.g. a SW-C shall have the possibility to write it to NV memory. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | 0 .. 65535 | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM493 :** | | |
| ***Name*** | NvmCrcNumOfBytes {NVM_CRC_NUM_OF_BYTES} | | |
| ***Description*** | If CRC is configured for at least one NVRAM block, this parameter defines the maximum number of bytes which shall be processed within one cycle of job processing. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | 1 .. 65535 | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module<br>dependency: CRC library | | |

| | |
|---|---|
| ***SWS Item*** | **NVM494 :** |
| ***Name*** | NvmDatasetSelectionBits {NVM_DATASET_SELECTION_BITS} |
| ***Description*** | Defines the number of least significant bits which shall be used to address a certain dataset of a NVRAM block within the interface to the memory hardware abstraction. 0..8: Number of bits which are used for dataset or redundant block addressing. 0: No dataset or redundant NVRAM blocks are configured at all, no selection bits required. 1: In case of redundant NVRAM blocks are configured, but no dataset NVRAM blocks. |

| | | | |
|---|---|---|---|
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | 0 .. 8 | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: ECU<br>dependency: MemHwA, NVM_NV_BLOCK_IDENTIFIER, NVM_BLOCK_MANAGEMENT_TYPE | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM495 :** | | |
| ***Name*** | NvmDevErrorDetect {NVM_DEV_ERROR_DETECT} | | |
| ***Description*** | Pre-processor switch to enable and disable development error detection. true: Development error detection enabled. false: Development error detection disabled. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM496 :** | | |
| ***Name*** | NvmDrvModeSwitch {NVM_DRV_MODE_SWITCH} | | |
| ***Description*** | Preprocessor switch to enable switching memory drivers to fast mode during performing NvM_ReadAll and NvM_WriteAll true: Fast mode enabled. false: Fast mode disabled. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM497 :** | | |
| ***Name*** | NvmDynamicConfiguration {NVM_DYNAMIC_CONFIGURATION} | | |
| ***Description*** | Preprocessor switch to enable the dynamic configuration management handling by the NvM_ReadAll request. true: Dynamic configuration management handling enabled. false: Dynamic configuration management handling disabled. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |

| | |
|---|---|
| ***Scope / Dependency*** | scope: module |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM498 :** | | |
| ***Name*** | NvmJobPrioritization {NVM_JOB_PRIORITIZATION} | | |
| ***Description*** | Preprocessor switch to enable job prioritization handling true: Job prioritization handling enabled. false: Job prioritization handling disabled. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM499 :** | | |
| ***Name*** | NvmMaxNoOfWriteRetries {NVM_MAX_NUM_OF_WRITE_RETRIES} | | |
| ***Description*** | Defines the maximum number of write retries for a NVRAM block with [NVM061]. Regardless of configuration a consistency check (and maybe write retries) are always forced for each block which is processed by the request NvM_WriteAll and NvM_WriteBlock. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | 0 .. 7 | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM500 :** | | |
| ***Name*** | NvmMultiBlockCallback {NVM_MULTI_BLOCK_CALLBACK} | | |
| ***Description*** | Entry address of the common callback routine which shall be invoked on termination of each asynchronous multi block request | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | FunctionNameDef | | |
| ***Default value*** | -- | | |
| ***regularExpression*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | |
|---|---|
| ***SWS Item*** | **NVM501 :** |
| ***Name*** | NvmPollingMode {NVM_POLLING_MODE} |
| ***Description*** | Preprocessor switch to enable/disable the polling mode in the NVRAM Manager and at the same time disable/enable the callback functions useable by lower layers true: Polling mode enabled, callback function usage disabled. false: Polling mode disabled, callback function usage enabled. |
| ***Multiplicity*** | 1 |
| ***Type*** | BooleanParamDef |

| | | | |
|---|---|---|---|
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM518_Conf :** | | |
| ***Name*** | NvmRepeatMirrorOperations {NVM_REPEAT_MIRROR_OPERATIONS} | | |
| ***Description*** | Defines the number of retries to let the application copy data to or from the NvM module's mirror before postponing the current job. | | |
| ***Multiplicity*** | 0..1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | 0 .. 7 | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM502 :** | | |
| ***Name*** | NvmSetRamBlockStatusApi {NVM_SET_RAM_BLOCK_STATUS_API} | | |
| ***Description*** | Preprocessor switch to enable the API NvM_SetRamBlockStatus. true: API NvM_SetRamBlockStatus enabled. false: API NvM_SetRamBlockStatus disabled. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM503 :** | | |
| ***Name*** | NvmSizeImmediateJobQueue {NVM_SIZE_IMMEDIATE_JOB_QUEUE} | | |
| ***Description*** | Defines the number of queue entries for the immediate priority job queue. If NVM_JOB_PRIORITIZATION is switched OFF this parameter shall be out of scope. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | 1 .. 65535 | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module<br>dependency: NVM_JOB_PRIORITIZATION | | |

| | |
|---|---|
| ***SWS Item*** | **NVM504 :** |
| ***Name*** | NvmSizeStandardJobQueue {NVM_SIZE_STANDARD_JOB_QUEUE} |
| ***Description*** | Defines the number of queue entries for the standard job queue. |

| | | | |
|---|---|---|---|
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | 1 .. 65535 | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM505 :** | | |
| ***Name*** | NvmVersionInfoApi {NVM_VERSION_INFO_API} | | |
| ***Description*** | Pre-processor switch to enable / disable the API to read out the modules version information [NVM285], [NVM286]. true: Version info API enabled. false: Version info API disabled. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| |
|---|
| ***No Included Containers*** |

**NVM028**: The following tables specify parameters that shall be definable in the module's configuration file (NvM_Cfg.h).

**NVM321**: Pre-compile time configuration parameters to be implemented as "`const`" should be placed into a separate c-file (NvM_Cfg.c).

### 10.2.4 NvmBlockDescriptor

| | |
|---|---|
| ***SWS Item*** | **NVM061 :** |
| ***Container Name*** | NvmBlockDescriptor |
| ***Description*** | Container for a management structure to configure the composition of a given NVRAM Block Management Type. Its multiplicity describes thenumber of configured NVRAM blocks, one block is required to be configured. The NVRAM block descriptors are condensed in the NVRAM block descriptor table. |
| ***Configuration Parameters*** | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM554 :** | | |
| ***Name*** | NvMRamBlockHeaderInclude | | |
| ***Description*** | Defines the header file where the owner of the NVRAM block has the declarations of the permanent RAM data block and/or ROM data block (if configured). If no permanent RAM block or ROM block is configured then this configuration parameter shall be ignored. | | |
| ***Multiplicity*** | 0..1 | | |
| ***Type*** | StringParamDef | | |
| ***Default value*** | -- | | |
| ***regularExpression*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: local | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM476 :** | | |
| ***Name*** | NvmBlockCRCType {NVM_BLOCK_CRC_TYPE} | | |
| ***Description*** | Defines CRC data width for the NVRAM block. Default: NVM_CRC16, i.e. CRC16 will be used if NVM_BLOCK_USE_CRC==TRUE | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | EnumerationParamDef | | |
| ***Range*** | NVM_CRC16 | (Default) CRC16 will be used if NVM_BLOCK_USE_CRC==TRUE. | |
| | NVM_CRC32 | CRC32 is selected for this NVRAM block. | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module<br>dependency: NVM_BLOCK_USE_CRC, NVM_CALC_RAM_BLOCK_CRC | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM477 :** | | |
| ***Name*** | NvmBlockJobPriority {NVM_BLOCK_JOB_PRIORITY} | | |
| ***Description*** | Defines the job priority for a NVRAM block (0 = Immediate priority). | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | 0 .. 255 | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |

| | | | |
|---|---|---|---|
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM062 :** | | |
| ***Name*** | NvmBlockManagementType {NVM_BLOCK_MANAGEMENT_TYPE} | | |
| ***Description*** | Defines the block management type for the NVRAM block.[NVM137] | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | EnumerationParamDef | | |
| ***Range*** | NVM_BLOCK_DATASET | NVRAM block is configured to be of dataset type. | |
| | NVM_BLOCK_NATIVE | NVRAM block is configured to be of native type. | |
| | NVM_BLOCK_REDUNDANT | NVRAM block is configured to be of redundant type. | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM036 :** | | |
| ***Name*** | NvmBlockUseCrc {NVM_BLOCK_USE_CRC} | | |
| ***Description*** | Defines CRC usage for the NVRAM block, i.e. memory space for CRC is reserved in RAM and NV memory. true: CRC will be used for this NVRAM block. false: CRC will not be used for this NVRAM block. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM519_Conf :** | | |
| ***Name*** | NvmBlockUseSyncMechanism {NVM_BLOCK_USE_SYNC_MECHANISM} | | |
| ***Description*** | Defines whether an explicit synchronization mechanism with a RAM mirror and callback routines for transferring data to and from NvM module's RAM mirror is used for NV block. true if synchronization mechanism is used, false otherwise. | | |
| ***Multiplicity*** | 0..1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | false | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | |
|---|---|
| ***SWS Item*** | **NVM033 :** |
| ***Name*** | NvmBlockWriteProt {NVM_BLOCK_WRITE_PROT} |
| ***Description*** | Defines an initial write protection of the NV block true: Initial block write protection is enabled. false: Initial block write protection is disabled. |
| ***Multiplicity*** | 1 |

| | | | |
|---|---|---|---|
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM119 :** | | |
| ***Name*** | NvmCalcRamBlockCrc {NVM_CALC_RAM_BLOCK_CRC} | | |
| ***Description*** | Defines CRC (re)calculation for the permanent RAM block or blocks configured to support explicit synchronization. true: CRC will be (re)calculated for this permanent RAM block. false: CRC will not be (re)calculated for this permanent RAM block. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module<br>dependency: NVM_BLOCK_USE_CRC | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM116 :** | | |
| ***Name*** | NvmInitBlockCallback {NVM_INIT_BLOCK_CALLBACK} | | |
| ***Description*** | Entry address of a block specific callback routine which shall be called if no ROM data is available for initialization of the NVRAM block. If a NULL pointer is configured, no specific callback routine shall be called for initialization of the NVRAM block with default data. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | FunctionNameDef | | |
| ***Default value*** | -- | | |
| ***regularExpression*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | |
|---|---|
| ***SWS Item*** | **NVM478 :** |
| ***Name*** | NvmNvBlockBaseNumber {NVM_NV_BLOCK_BASE_NUMBER} |
| ***Description*** | Configuration parameter to perform the link between the NVM_NVRAM_BLOCK_IDENTIFIER used by the SW-Cs and the FEE_BLOCK_NUMBER expected by the memory abstraction modules. The parameter relates directly to the FEE_BLOCK_NUMBER or EA_BLOCK_NUMBER with all configured NVM_DATASET_SELECTION_BITS set to zero (ref. to chapter 7.1.2.1). Calculation Formula: "value = TargetBlockReference.[Ea/Fee]BlockConfiguration.[Ea/Fee]BlockNumber" |
| ***Multiplicity*** | 1 |
| ***Type*** | DerivedIntegerParamDef |

| | | | |
|---|---|---|---|
| ***Range*** | .. | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module<br>dependency: FEE_BLOCK_NUMBER, EA_BLOCK_NUMBER | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM479 :** | | |
| ***Name*** | NvmNvBlockLength {NVM_NV_BLOCK_LENGTH} | | |
| ***Description*** | Defines the NV block data length in bytes | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | 1 .. 65535 | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM480 :** | | |
| ***Name*** | NvmNvBlockNum {NVM_NV_BLOCK_NUM} | | |
| ***Description*** | Defines the number of multiple NV blocks in a contiguous area according to the given block management type. 1-255 For NVRAM blocks to be configured of block management type NVM_BLOCK_DATASET. The actual range is limited according to NVM148. 1 For NVRAM blocks to be configured of block management type NVM_BLOCK_NATIVE 2 For NVRAM blocks to be configured of block management type NVM_BLOCK_REDUNDANT | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | 1 .. 255 | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module<br>dependency: NVM_BLOCK_MANAGEMENT_TYPE | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM481 :** | | |
| ***Name*** | NvmNvramBlockIdentifier {NVM_NVRAM_BLOCK_IDENTIFIER} | | |
| ***Description*** | Identification of a NVRAM block via a unique block identifier. Implementation Type: NvM_BlockIdType. min = 0 max = 2^(16-NVM_DATASET_SELECTION_BITS)-1 Reserved NVRAM block IDs: 0 -> to derive multi block request results via NvM_GetErrorStatus 1 -> redundant NVRAM block which holds the configuration ID | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef (Symbolic Name generated for this parameter) | | |
| ***Range*** | 0 .. | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module<br>dependency: NVM_DATASET_SELECTION_BITS | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM035 :** | | |
| ***Name*** | NvmNvramDeviceId {NVM_NVRAM_DEVICE_ID} | | |
| ***Description*** | Defines the NVRAM device ID where the NVRAM block is located. Calculation Formula: "value = TargetBlockReference.[Ea/Fee]BlockConfiguration.[Ea/Fee]DeviceIndex" | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | DerivedIntegerParamDef | | |
| ***Range*** | .. | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module<br>dependency: EA_DEVICE_INDEX, FEE_DEVICE_INDEX | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM482 :** | | |
| ***Name*** | NvmRamBlockDataAddress {NVM_RAM_BLOCK_DATA_ADDRESS} | | |
| ***Description*** | Defines the start address of the RAM block data. If a NULL pointer is configured, no permanent RAM data block is available for the selected block management type. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | StringParamDef | | |
| ***Default value*** | -- | | |
| ***regularExpression*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM521_Conf :** | | |
| ***Name*** | NvmReadRamBlockFromNvCallback {NVM_READ_RAM_BLOCK_FROM_NVM} | | |
| ***Description*** | Entry address of a block specific callback routine which shall be called in order to let the application copy data from the NvM module's mirror to RAM block. Implementation type: Std_ReturnType E_OK: copy was successful E_NOT_OK: copy was not successful, callback routine to be called again | | |
| ***Multiplicity*** | 0..1 | | |
| ***Type*** | FunctionNameDef | | |
| ***Default value*** | -- | | |
| ***regularExpression*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | |
|---|---|
| ***SWS Item*** | **NVM483 :** |
| ***Name*** | NvmResistantToChangedSw {NVM_RESISTANT_TO_CHANGED_SW} |
| ***Description*** | Defines whether a NVRAM block shall be treated resistant to configuration changes or not. If there is no default data available at configuration time then the application shall be responsible for providing the default initialization data. In this |

| | | | |
|---|---|---|---|
| | case the application has to use NvM_GetErrorStatus()to be able to distinguish between first initialization and corrupted data. true: NVRAM block is resistant to changed software. false: NVRAM block is not resistant to changed software. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM484 :** | | |
| ***Name*** | NvmRomBlockDataAddress {NVM_ROM_BLOCK_DATA_ADDRESS} | | |
| ***Description*** | Defines the start address of the ROM block data. If a NULL pointer is configured, no ROM block is available for the selected block management type. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | StringParamDef | | |
| ***Default value*** | -- | | |
| ***regularExpression*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM485 :** | | |
| ***Name*** | NvmRomBlockNum {NVM_ROM_BLOCK_NUM} | | |
| ***Description*** | Defines the number of multiple ROM blocks in a contiguous area according to the given block management type. 0-255 For NVRAM blocks to be configured of block management type NVM_BLOCK_DATASET. The actual range is limited according to NVM148. 0-1 For NVRAM blocks to be configured of block management type NVM_BLOCK_NATIVE 0-1 For NVRAM blocks to be configured of block management type NVM_BLOCK_REDUNDANT | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | IntegerParamDef | | |
| ***Range*** | .. | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module<br>dependency: NVM_BLOCK_MANAGEMENT_TYPE, NVM_NV_BLOCK_NUM | | |

| | |
|---|---|
| ***SWS Item*** | **NVM117, NVM245 :** |
| ***Name*** | NvmSelectBlockForReadall {NVM_SELECT_BLOCK_FOR_READALL} |
| ***Description*** | NVM117: Defines whether a NVRAM block shall be processed during NvM_ReadAll or not. This configuration parameter has influence on those NVRAM blocks which are configured to have a permanent RAM block or NVRAM blocks which are configured to have explicit synchronization mechanism with a RAM mirror and callback routines. NVM245: Blocks of management type NVM_BLOCK_DATASET shall not be loaded automatically upon start-up. Thus |

| | | | |
|---|---|---|---|
| | the selection of blocks, which belong to block management type NVM_BLOCK_DATASET, shall not be possible for the service NvM_ReadAll. true: NVRAM block shall be processed by NvM_ReadAll false: NVRAM block shall not be processed by NvM_ReadAll | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module<br>dependency: NVM_RAM_BLOCK_DATA_ADDRESS | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **:** | | |
| ***Name*** | NvmSingleBlockCallback {NVM_SINGLE_BLOCK_CALLBACK} | | |
| ***Description*** | Entry address of the block specific callback routine which shall be invoked on termination of each asynchronous single block request [NVM113]. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | FunctionNameDef | | |
| ***Default value*** | -- | | |
| ***regularExpression*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM072 :** | | |
| ***Name*** | NvmWriteBlockOnce {NVM_WRITE_BLOCK_ONCE} | | |
| ***Description*** | Defines write protection after first write. The NVRAM manager sets the write protection bit after the NV block was written the first time. This means that some of the NV blocks in the NVRAM should never be erased nor be replaced with the default ROM data after first initialization. [NVM276]. true: Defines write protection after first write is enabled. false: Defines write protection after first write is disabled. | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | BooleanParamDef | | |
| ***Default value*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| | |
|---|---|
| ***SWS Item*** | **NVM520_Conf :** |
| ***Name*** | NvmWriteRamBlockToNvCallback {NVM_WRITE_RAM_BLOCK_TO_NVM} |
| ***Description*** | Entry address of a block specific callback routine which shall be called in order to let the application copy data from RAM block to NvM module's mirror. Implementation type: Std_ReturnType E_OK: copy was successful E_NOT_OK: copy was not successful, callback routine to be called again |

| | | | |
|---|---|---|---|
| ***Multiplicity*** | 0..1 | | |
| ***Type*** | FunctionNameDef | | |
| ***Default value*** | -- | | |
| ***regularExpression*** | -- | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | X | All Variants |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | scope: module | | |

| ***Included Containers*** | | |
|---|---|---|
| ***Container Name*** | ***Multiplicity*** | ***Scope / Dependency*** |
| NvmTargetBlockReference | 1 | -- |

### 10.2.5 NvmTargetBlockReference

| | |
|---|---|
| ***SWS Item*** | **NVM486 :** |
| ***Choice container Name*** | NvmTargetBlockReference |
| ***Description*** | -- |

| ***Container Choices*** | | |
|---|---|---|
| ***Container Name*** | ***Multiplicity*** | ***Scope / Dependency*** |
| NvmEaRef | 0..1 | EEPROM Abstraction |
| NvmFeeRef | 0..1 | Flash EEPROM Emulation |

### 10.2.6 NvmEaRef

| | |
|---|---|
| ***SWS Item*** | **NVM487 :** |
| ***Container Name*** | NvmEaRef |
| ***Description*** | EEPROM Abstraction |
| ***Configuration Parameters*** | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM488 :** | | |
| ***Name*** | NvmNameOfEaBlock | | |
| ***Description*** | reference to EaBlock | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | Reference to [ EaBlockConfiguration ] | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | -- | |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | | | |

| ***No Included Containers*** |
|---|

### 10.2.7 NvmFeeRef

| | |
|---|---|
| ***SWS Item*** | **NVM489 :** |
| ***Container Name*** | NvmFeeRef |
| ***Description*** | Flash EEPROM Emulation |
| ***Configuration Parameters*** | |

| | | | |
|---|---|---|---|
| ***SWS Item*** | **NVM490 :** | | |
| ***Name*** | NvmNameOfFeeBlock | | |
| ***Description*** | reference to FeeBlock | | |
| ***Multiplicity*** | 1 | | |
| ***Type*** | Reference to [ FeeBlockConfiguration ] | | |
| ***ConfigurationClass*** | ***Pre-compile time*** | -- | |
| | ***Link time*** | -- | |
| | ***Post-build time*** | -- | |
| ***Scope / Dependency*** | | | |

| |
|---|
| ***No Included Containers*** |

## 10.3 Common configuration options

**NVM030:** By use of configuration techniques, each application shall be enabled to declare the memory requirements at configuration time. This information shall be useable to assign memory areas and to generate the appropriate interfaces. Wrong memory assignments and conflicts in requirements (sufficient memory not available) shall be detected at configuration time.

**NVM034:** The NVRAM memory layout configuration shall have a unique ID. The NvM module shall have a configuration identifier that is a unique property of the memory layout configuration. The ID can be either statically assigned to the configuration or it can be calculated from the configuration properties. This should be supported by a configuration tool. The ID must be changed if the block configuration changes, i.e. if a block is added or removed, or if its size or type is changed. The ID shall be stored together with the data and shall be used in addition to the data checksum to determine the consistency of the NVRAM contents.

**NVM073:** The comparison between the stored configuration ID and the compiled configuration ID shall be done as the first step within the function `NvM_ReadAll` [NVM008] during startup. In case of a detected configuration ID mismatch, the behavior of the NvM module shall be defined by a configurable option [NVM028].

**NVM052:** Provide information about used memory resources. The NvM module configuration shall provide information on how many resources of RAM, ROM and NVRAM are used. The configuration tool shall be responsible to provide detailed information about all reserved resources. The format of this information shall be commonly used (e.g. MAP file format).

### 10.3.1 Published parameters

Published information contains data defined by the implementer of the SW module that does not change when the module is adapted (i.e. configured) to the actual HW/SW environment. It thus contains version and manufacturer information.

The standard common published information like

vendorId (<Module>_VENDOR_ID),
moduleId (<Module>_MODULE_ID),
arMajorVersion (<Module>_AR_MAJOR_VERSION),
arMinorVersion (<Module>_ AR_MINOR_VERSION),
arPatchVersion (<Module>_ AR_PATCH_VERSION),
swMajorVersion (<Module>_SW_MAJOR_VERSION),
swMinorVersion (<Module>_ SW_MINOR_VERSION),
swPatchVersion (<Module>_ SW_PATCH_VERSION),

vendorApiInfix (<Module>_VENDOR_API_INFIX)

is provided in the BSW Module Description Template (see 3.1 Figure 4.1 and Figure 7.1).

Additional published parameters are listed below if applicable for this module.

# 11 AUTOSAR Service implemented by the NVRAM Manger

## 11.1 Scope of this Chapter

This chapter is an addition to the specification of the NvM module. Whereas the other parts of the specification define the behavior and the C-interfaces of the corresponding basic software module, this chapter formally specifies the corresponding AUTOSAR Service in terms of the SWC Template. The interfaces described here will be visible on the VFB and are used to generate the RTE between application software and the NvM module.

## 11.2 Overview

### 11.2.1 Architecture

In the AUTOSAR ECU Architecture (see [1]) the NvM module implements an AUTOSAR Service as indicated in **Figure 19**.

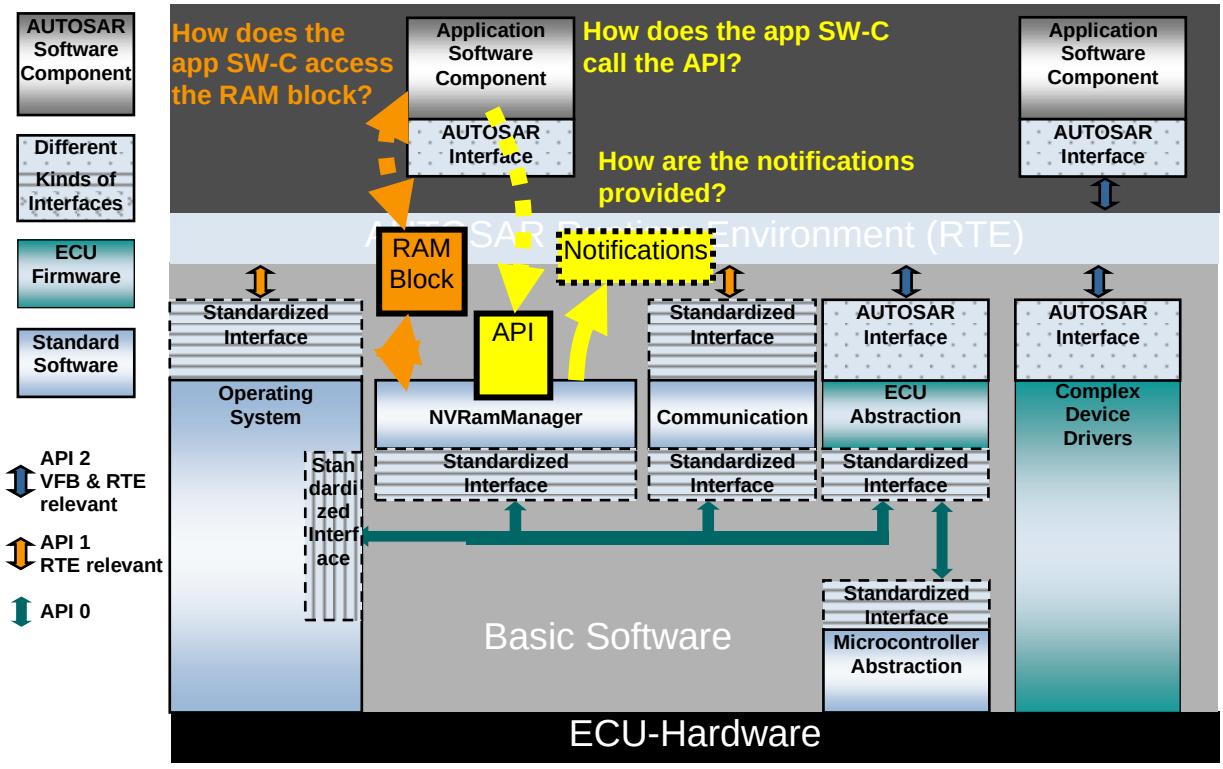

**Figure 19: NVRAM Manager in ECU software architecture**

From the viewpoint of the basic software C-module “NVRAM Manager”, there are three kinds of dependencies between the Service and the AUTOSAR Software Components above the RTE[2] :

- the application accesses the API (implemented as C-functions) of the NvM module
- the application is optionally notified upon the outcome of requested asynchronous activity (via callback-C-functions by the NvM module)
- the application accesses the RAM-blocks which the NvM module has to save or restore.

These dependencies must be described in terms of the AUTOSAR meta-model which will contribute to the SW-C Description of the application component as well as to the SW-C Description of the NVRAM Service.

### 11.2.2 Requirements

There are three sources of requirements for this specification:

- The requirements for the functionality of the NVRAM service are specified in [3].
- In order to model the VFB view of the Service, the chapter on AUTOSAR Services of the VFB specification [7] has to be considered as an additional requirement.
- For the formal description of the SW-C attributes [8] gives the requirements.

### 11.2.3 Use Cases

On each ECU we have typically one instance of the NVRAM Service and several Atomic Software Component instances, named “clients” further on in this chapter, which are using this Service. In addition, there are parts of the basic software, which either control the NvM module (e.g. for initiation and shutdown) or need to read or write some NVRAM data themselves.

Each client will own certain transient data which it “wants” to make persistent via the NVRAM Service. It is important to mention that the transient data are privately owned by clients[3] – otherwise, the clients could communicate via the content of those data which is against AUTOSAR principles. Towards the NVRAM Service, the client uses so-called block identifiers to address the persistent data.

Furthermore, it is important that in general a client is a component *instance*. So if a client component can be instantiated more than once on an ECU, each instance will in general possess its own private copy of transient data. This is an important restriction on the modeling of block identifiers: Because for a reusable component,

[2] “Applications” of the NVRAM Manager can be “below” as well as “above” the RTE; this chapter concentrates on the interfaces seen from the applications above the RTE.

[3] If the client is an Atomic SWC. It does not hold for components of the basic SWC which might use the NVRAM Manager.

we cannot a priori prescribe the number of instances. We can use block identifiers (which could be in the form of symbolic information) in the code only on a per-component base, but this must result in different actual identifiers towards the NVRAM.

There are three principle ways, how a client can use the NVRAM service.

#### 11.2.3.1 Implicit Update and Restore of RAM Mirror

See **Figure 20**. This is a summary of use cases in which there is no communication (in the sense of the VFB) between the NvM module and its client. The client holds its transient data in a so-called RAM mirror which is organized in so-called RAM blocks permanently associated with their non-volatile counterparts of the NvM module. Update and restore of the RAM blocks is initiated by the ECU State Manager, the AUTOSAR SW-C accesses its RAM blocks directly. Note that the NvM module does NOT communicate with its client via the RAM blocks because no information is exchanged between the NvM module and its client via the RAM block.
In this case, the client does not need any AUTOSAR Interface to communicate with the Service. But nevertheless, there have to be certain contracts between the client and the NvM module:

- On the size, identification and maybe further properties of the RAM/NVRAM blocks which have to be known on both sides. For the access of the RAM mirror via the RTE see 11.4.
- On the states (of the ECU and/or the client component) in which the RAM blocks can be safely accessed by the client. For this contract, it is essential that the client is informed about its activation/termination or about the relevant ECU states. This communication path has to be handled by the RTE in cooperation with the ECU State Manager (not visible in **Figure 20**).

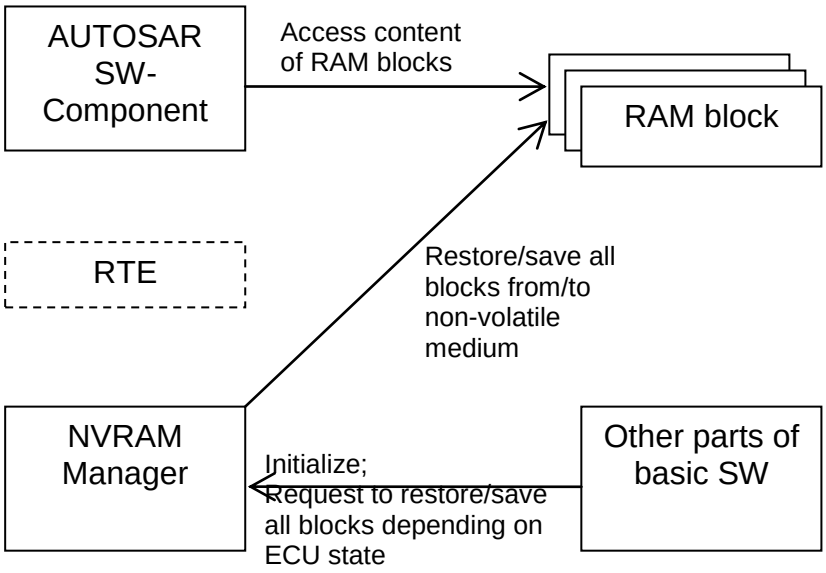


**Figure 20: Implicit update/restore of RAM blocks**

#### 11.2.3.2 Explicit Update and Restore of RAM Mirror

See **Figure 21**. This is a summary of use cases in which the client explicitly requests to save or restore the content of a mirror RAM block from/to non-volatile memory (e.g. in order to decrease the risk of data loss). But as in the first case, RAM blocks are permanently associated with their counterparts in NVRAM. Now we need direct communication (in the sense of the VFB) between the NvM module and its client. Also in this case, the NvM module does NOT communicate with its client via the RAM blocks. In this case, the client must use an AUTOSAR interface to communicate with the service.

Similar to the case of implicit update/restore, we need certain contracts between the client and the NvM module:

- On the size, identification and maybe further properties of the RAM/NVRAM blocks which have to be known on both sides. For the access of the RAM mirror via the RTE see 11.4.
- On the states (of the ECU and/or the client component) in which the client may request to restore or save RAM blocks. For this contract, it is essential that the client is informed about its activation/termination or relevant ECU states. This communication path has to be handled by the RTE in cooperation with the ECU State Manager (not visible in **Figure 21**).
- During the explicit update and restore scenarios, the client has to follow certain rules, e.g. not to access a RAM block before the update or restore is finished.

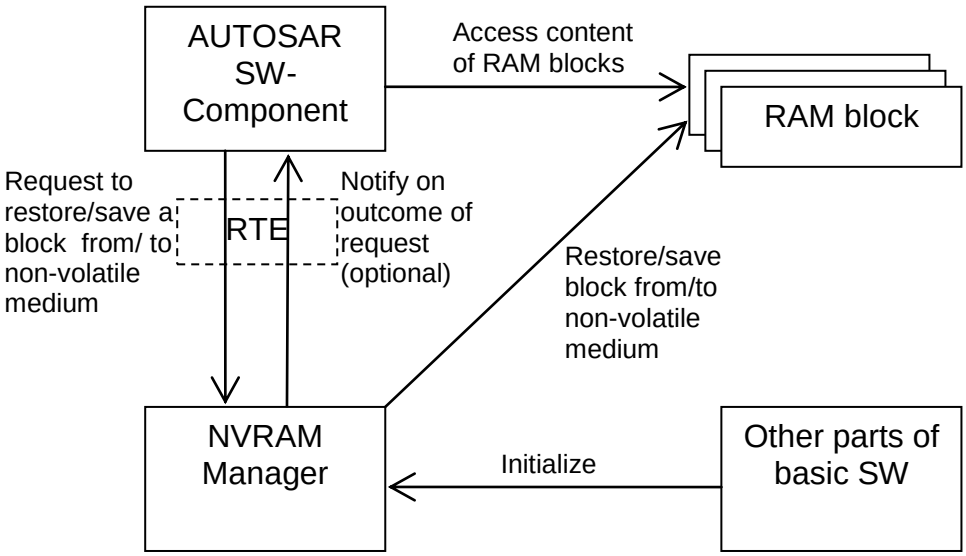


**Figure 21: Explicit update/restore of mirror RAM blocks**

#### 11.2.3.3 Explicit Update and Restore via a Local Buffer

See **Figure 22**. This is a summary of use cases in which the client explicitly requests to save or restore the content of some transient data, but instead of using a RAM mirror, it passes the address of a buffer which is only temporarily associated with a certain block in NVRAM. Also, we need here direct communication (in the sense of the VFB) between the NvM module and its client. Also in this case, the NvM module does NOT communicate with its client via the RAM blocks.

Also in this case, the client must use an AUTOSAR interface to communicate with the service.

Again we need certain contracts between the client and the NvM module:

- On the size, identification and maybe further properties of the temporarily associated NVRAM blocks which have to be known on both sides. For the buffer, the partners do not have to agree on its allocation because it is totally up to the client.
- On the states (of the ECU and/or the client component) in which the client may request to restore or save RAM blocks. For this contract, it is essential that the client is informed about its activation/termination or relevant ECU states. This communication path has to be handled by the RTE in cooperation with the ECU State Manager (not visible in **Figure 21**).
- During the explicit update and restore scenarios, the client has to follow certain rules, e.g. not to access the buffer before the update or restore is finished.

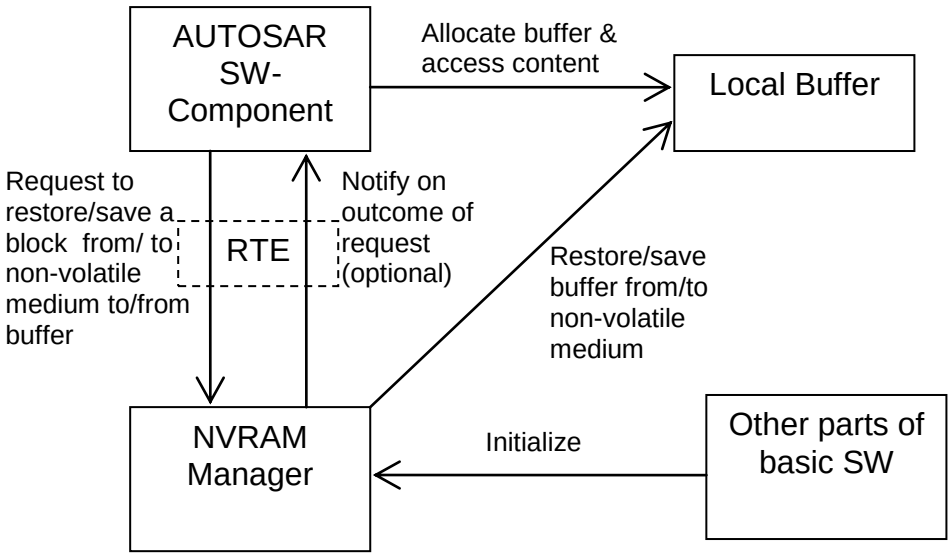


**Figure 22: Explicit update/restore of RAM blocks via buffer**

## 11.3 Specification of the Ports and Port Interfaces

This chapter specifies the ports and port interfaces which are needed in order to operate the NvM module functionality over the VFB. Note that there are ports on both sides of the RTE: The SW-C description of the NVRAM Service defines the ports below the RTE. Each SW-Component which uses the Service must contain “service ports” in its own SW-C description which are connected to the ports of the NvM module, so that the RTE can be generated.

### 11.3.1 Ports and Port Interface for Single Block Requests

#### 11.3.1.1 General Approach

It is appropriate to model the requests issued from a client to the NVRAM Service by ports with client/server interfaces.

Typically, a client of the application domain needs the NvM module for services dealing with individual blocks (except for the simple use case described in 11.2.3.1). These so-called single block requests of the NvM module C-API need the Block ID as a first argument for the C-function.

In order to keep the client code independent from the configuration of NVRAM block IDs (which depend on the needs of all the applications on an ECU), the block IDs are not passed from the clients to the NvM module, but are modeled as “port defined argument values” of the Provide Ports on the NvM module side. As a consequence, the block IDs will not show up as arguments in the operations of the client-server interface. As a further consequence of this approach, there will be separate ports for each NVRAM block both on the client side as well as on the server side.

#### 11.3.1.2 Data Types

This chapter describes the data types which will be used in the port interfaces for single block requests and notifications.

The data types `uint8` and `boolean` used in the interfaces refer to the basic AUTOSAR data types.

The data type `RequestResultType` indicates the result of a read or write request which are defined by the SW-C Description as follows:

```
IntegerType RequestResultType {
     upperLimit = 0
     lowerLimit = 5
};

Via an associated CompuMethod the following constants are
defined within this type:
     0 -> NVM_REQ_OK
```

```
1 -> NVM_REQ_NOT_OK
2 -> NVM_REQ_PENDING
3 -> NVM_INTEGRITY_FAILED
4 -> NVM_REQ_BLOCK_SKIPPED
5 -> NVM_REQ_NV_INVALIDATED
```

For the use cases described in chapter 11.2.3.3, we need a type `DstPtrType` to pass the address of the local buffer:

```
ArrayType DstPtrType
{
      elementType = uint8;
      maxNumberOfElements = <xx>;
};
```

Where <xx> denotes the size of the used buffer, which must be equal or bigger than the size of the data block associated with the port interface in which the buffer is used.

This solution may result in slightly different interfaces if different buffer sizes are used. This is the only possibility to describe the situation in the current SWCT because it is not possible to specify a pointer type. It is assumed that for operations which can be invoked concurrently, the RTE will generate just a pointer (of type uint*) and allocates no additional buffer so that the resulting C-code will be as efficient as a direct function call passing a pointer.

Note that for read/write operations which do not need a local buffer, the client has to pass a zero pointer instead of the buffer address.

#### 11.3.1.3 Port Interface

All single block operations (with the exception of SetBlockProtection which is discussed in chapter 11.3.3) are put into one single port interface in order to minimize the number of ports and names needed in the XML description.

The operations correspond to the function calls of the NVRAM C-API (notation in pseudo code; must be transferred into XML). Compared to the C-function, we do not need the “NvM_” prefix in the names because the names given here will show up in the XML not as global entities but as part of an interface description.

The notation of possible error codes resulting from server calls follows the approach in the meta-model. It is a matter of the RTE specification [9] how those error codes will be passed via the actual API.

```
ClientServerInterface NvMService {
      PossibleErrors {
            E_NOT_OK = 1
      };

      // the next operation is always provided
      GetErrorStatus( OUT RequestResultType RequestResultPtr );
```

```
        // the next two operations are always available, but are needed
        // only for "data set" block types. Thus they shall be provided in
        // the interface only for those block types
        SetDataIndex( IN uint8 DataIndex );
        GetDataIndex( OUT uint8 DataIndexPtr );

        // this operation is only provided via optional configuration
        // NvmSetRamBlockStatusApi
        SetRamBlockStatus( IN boolean BlockChanged );

        // the next three operations are only provided for
        // NVRAM API configuration class 2 and 3
        ReadBlock( OUT DstPtrType DstPtr, ERR{E_NOT_OK} );
        WriteBlock( IN DstPtrType SrcPtr, ERR{E_NOT_OK} );
        RestoreBlockDefaults( OUT DstPtrType DstPtr, ERR{E_NOT_OK} );

        // the next two operations are only provided for
        // NVRAM API configuration class 3
        EraseBlock( ERR{E_NOT_OK} );
        InvalidateNvBlock( ERR{E_NOT_OK} );
};
```

Note that in the interface of each Require Port of the client, only those operations must be present which the client actually requires for that block.
On the other hand, in the interface of the Provide Port, only those operations will be provided, which actually have to be configured for the NVRAM Service on a specific ECU. Because some of the operations are optional, they may not always be provided. The optional parts are indicated by comments in the pseudo-code and must currently be configured manually in the XML Specification[4].

Note that via compatibility rules, a Require Port can be connected to a Provide Port providing more operations than actually needed, but not the other way round.

Note also, that due to the optional parts, different variants of this interface may exist on the same ECU. In this case, it may be required to make the interface name unique by attaching suffixes to its name. A rule for this is however not standardized.

#### 11.3.1.4 Ports

**Figure 23** shows how AUTOSAR Software components (single or multiple instances) are connected by means of service ports to the NvM module.

[4] By introducing a property concept in the meta-model, the optional parts may in future be controlled by other model elements. For this, the relation between the Service requirements of an SWC and the Service ports of an SWC must be explicitly modeled.

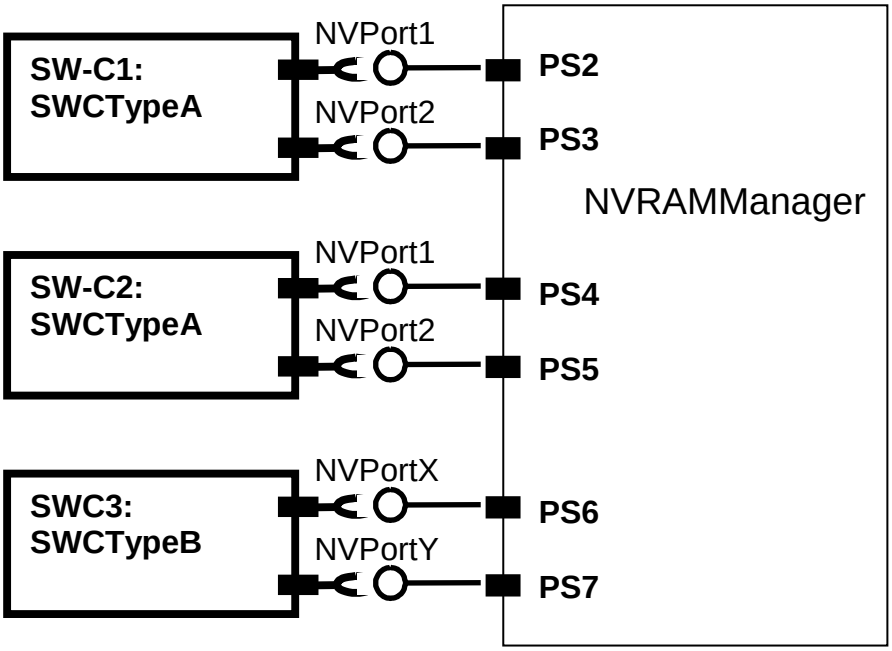


**Figure 23: Example of SW-Cs connected to the NVRAM via service ports. On the left side, there are two instances of component SWCTypeA and one instance of component SWCTypeB. The Port names on the left side are only for illustration. No notification ports or administrative ports are configured.**

On the NvM module side, there is one port per NVRAM block providing all the services of the interface `NvMService` described above. Each client has one port for requiring those services for each NVRAM block associated with that client.
The ports providing the services are named according to the `BlockID`:
`PS2, PS3, … ,PS7`

These names are examples, they are not standardized. It is not essential, that these ports are numbered consecutively, but the numbers should match the NVRAM blocks for documentation purposes.

*Hint for the developers of the client components:*

There are two ways of accessing the NvM Service ports via the RTE. The selection of one alternative is an implementation decision independent of the specification of the NVRAM Service.

1) Use the “direct API” of the RTE. In the client code, each single block operation will show up as a separate call to the RTE which can be optimized to “zero overhead” function invocation under certain conditions (source code of the client is available and the client is a single-instance component). In case of optimization, there will be a macro for each operation and each block. Each macro will be expanded to a direct call of the NvM module.
2) Use the “indirect API” of the RTE. The client code will call the single block operations via a “port handle”. This port handle can be accessed by an array index which in this case identifies the NVRAM blocks used by this client. This approach offers less potential for code size optimization in the RTE, but the port

handle allows for a more compact handling in the client code, e.g. if iteration over NVRAM blocks is required. Note that the ports accessed via this kind of API must have the same port interface, i.e. the optional parts must be the same.

For more explanation of the direct and indirect API see [9].

### 11.3.2 Ports and Port Interface for Notifications

Some block requests which can be initiated by client requests are performed asynchronously (with respect to the requesting call) and can be configured to throw “notifications” to indicate the completion (or error) of such a request. The reason is the relatively slow access time of the storage medium.

As a first glance a pair of request/notification looks like an asynchronous client/server communication described in [7], chapter 4.1.7.2. But it cannot be modeled that way, because in an asynchronous client/server communication, the server operation in itself is always synchronous whereas for the NvM module, the request of an action, the actual performing of it and the notification are asynchronous activities. Another reason is that the notifications are optional.

Therefore it is required to model the notification by a separate connection and thus separate ports on both service and SW component side.

The NvM module on C-code level does not pass a block ID with the callback but allows configuring a separate callback address for each block. This means that on the modeling level we need one separate (optional) notification port for each single block. As a consequence, the existence of notification ports in the SW-C Description of the NVRAM Service must be configured per ECU.

For the notification ports, a client/server interface is used, because we have to transmit two values. A drawback of this approach (in comparison to sender/receiver) is that a badly implemented SW-C could block the NvM module for an unacceptable amount of time. To minimize this risk there are two possibilities:

- Design rules for the implementation of such notifications by the SW-Cs
- Configuring the notification interface as asynchronous client/server would allow decoupling of the invocation in the SW-C from the implementation of the notification function. As this involves a more complicated protocol, this shall however be used only in special cases and is not part of the standardization of this Service.

The notification has to pass the following arguments:

- A “Service ID” which indicates which one of the asynchronous services triggered via the operations of Interface `NvMService` (see above) the notification belongs to.
- A `JobRequestResult` indicating success or failure of the service.

The Pseudo-Code for the notification interface (which has to be translated into XML) is:

```
ClientServerInterface NvMNotifyJobFinished {
      JobFinished( IN uint8 ServiceId,
                   IN RequestResultType JobResult);

};
```

The Pseudo-Code for the init block notification interface (which has to be translated into XML) is:

```
ClientServerInterface NvMNotifyInitBlock
{
InitBlock();

};
```

It is recommended to name the ports providing the notifications via this interface according to the `BlockID` (but this is no standard), for example:
`PN2, PN3,…`

Interrupt context: „This routine might be called in interrupt context, depending on the calling function.“ (see Chapter 0).

This requirement is consistent with current RTE requirements if the interrupt context is not propagated ”above“ the RTE. This must be handled by the RTE generation. Currently (in AUTOSAR 2.1), it is not possible to indicate this condition in the SWC Description of the Service. As a workaround, the runnable implementing the callback in the client component must be marked in this case as “canBeInvokedConcurrently = FALSE”. Note that this is only a workaround because this condition should be handled independently of the SWC description of the client components.

### 11.3.3 Ports and Port Interfaces for Administrative Operations

Administrative functions which are not needed for “normal” use cases are put into a separate port interface. Currently it contains only the operation “SetBlockProtection”.

The port interface is defined as follows. It does not specify error codes:

```
ClientServerInterface NvMAdministration {

      // the next operation is only provided for NVRAM API configuration
      // class 3; besides of that it should only be required by a client
      // for special use cases, e.g. if a block protection has to be
      // removed during initiating of EOL data
      SetBlockProtection( IN boolean ProtectionEnabled );
};
```

For the purpose of this document, ports providing this interface have the name PAdmin<nn> for a port identifier <nn>.

### 11.3.4 Ports and Port Interfaces for Mirror Operations

The mirror functions provided by the client (to be called by the NvM module) and described in Chapter 7.2.2.17 are summarized here:

**NVM738:**

```
ClientServerInterface NvMMirror {
 PossibleErrors {
  E_NOT_OK = 1
 };
   ReadRamBlockFromNvm ( IN void DATA_REFERENCE SrcPtr, ERR{E_NOT_OK} );
   WriteRamBlockToNvm ( IN void DATA_REFERENCE DstPtr, ERR{E_NOT_OK} );
}
```

### 11.3.5 Summary of all Ports

We end up with the following structure for the AUTOSAR Interface of the NvM module:

- For access from “normal” application components above the RTE:
  - For each memory block:
    - One port with a client-server interface providing all block-related services for that block (except SetBlockProtection).
    - Optional: A port with a client-server interface requiring a callback related to notifications for that block.
- For administrative purposes from above the RTE:
  - For each memory block:
    - Optional: One port with a client-server interface providing administrative services for that block

We indicate this as follows in pseudo code:

```
Service NvM
{
      // the entries in the next section will be defined only, if they
      // are actually required by client service ports
      ProvidePort NvMService PS2;
      ProvidePort NvMService PS3;
      ProvidePort NvMService PS4;
      …
      // the entries in the next section will be defined only, if the
      // notification sinks are actually provided by client service ports
      RequirePort NvMNotify PN2;
      RequirePort NvMNotify PN3;
      RequirePort NvMNotify PN4;
      …
      // the entries in the next section will be defined only, if they
      // are actually required by client service ports, i.e. in special
      // cases, where SetBlockProtection is needed
      ProvidePort NvMAdmin PAdmin2;
      ProvidePort NvMAdmin PAdmin3;
      ProvidePort NvMAdmin PAdmin4;
    …
      // the entries in the next section will be defined only, in case of
      // explicit synchronization
      RequirePort NvMMirror PM2;
      RequirePort NvMMirror PM3;
      RequirePort NvMMirror PM4;
      …

};
```

It is obvious that the existence of all these port definitions depends on the ECU. But also the port interfaces itself are in general ECU dependent because they contain optional parts as shown before. But they consist of standardized elements (operation prototypes, data types).

## 11.4 Access to the Memory Blocks

The NvM module does not specify how its clients actually access the RAM block. This is the private responsibility of each client. This especially holds true for the structure and semantics of the block content.

However, the current meta model allows the SW-C to specify the overall features of its NVRAM blocks (identifier, size, criticality, existence of ROM defaults etc..). For details see the specification of PortAnnotation. The NVRAM service has to be configured in response to those configurations given in the SW-C descriptions of its clients. The configuration parameters of the NVRAM Service are however more concrete (it has for example a “block management type”) whereas the SW-C template defines the requirements on such a service in a more feature-related way.

The following is only relevant if the client uses a “RAM mirror” (use cases 11.2.3.1 and 11.2.3.2): From the viewpoint of the client, each block of its RAM mirror shall be modeled as a PerInstanceMemory. The RTE will then handle static allocation of

those memory regions. In order to get access to one of its memory blocks in RAM, the client has to use the RTE API: “Rte_Pim” (see [9]).

By this, the client gets a correctly typed handle to the memory section.There is no need to standardize the name of the memory section because the RTE expands it with the component and/or instance name in order to avoid name clashes.

Note the start address and size of the C data object associated with the PerInstanceMemory sections must be harmonized with the configuration of the NVRAM block descriptors in the ECU configuration description. This issue is not part of this Specification.

## 11.5 InternalBehavior

The NvM module specification does not standardize the basic type to be used for identifying the NVRAM blocks since the needed binary size is ECU dependent. This type has to be defined for a specific ECU as follows:

```
IntegerType BlockIdType {
          lowerLimit = 2;
          upperLimit = <xx>;
     };

// Where <xx> = 2^(16- NVM_DATASET_SELECTION_BITS)-1
// see NVMRAM manager SWS for explanation)
```

This type does not show up in the service ports of the client components because the block identifier is implemented as port defined argument value (see chapter 11.5) which is part of the InternalBehavior of the NVRAM Service. So the ECU dependency of BlockIdType is not visible for the clients.

Values 0 and 1 have special internal meaning and must not be used as identifiers for “normal” NVRAM blocks.

The InternalBehavior of the NVRAM Service is only seen by the local RTE. Besides the definition of the block identifiers as port defined arguments, it must specify the operation invoked runnables:

```
InternalBehavior NVRAMManager {

      // definition of associated operation-invoked RTE-events not shown
      // (it is done in the same way as for any SWC type)

      // section “runnable entities”:
      RunnableEntity GetErrorStatus
            symbol “NvM_GetErrorStatus”
            canbeInvokedConcurrently = TRUE

      RunnableEntity SetDataIndex
            symbol “NvM_SetDataIndex”
```

```
            canbeInvokedConcurrently = TRUE

      RunnableEntity GetDataIndex
            symbol "NvM_GetDataIndex"
            canbeInvokedConcurrently = TRUE

      RunnableEntity SetRamBlockStatus
            symbol "NvM_SetRamBlockStatus"
            canbeInvokedConcurrently = TRUE

      RunnableEntity ReadBlock
            symbol "NvM_ReadBlock"
            canbeInvokedConcurrently = TRUE

      RunnableEntity WriteBlock
            symbol "NvM_WriteBlock"
            canbeInvokedConcurrently = TRUE

      RunnableEntity RestoreBlockDefaults
            symbol "NvM_RestoreBlockDefaults"
            canbeInvokedConcurrently = TRUE

      RunnableEntity EraseNvBlock
            symbol "NvM_EraseNvBlock"
            canbeInvokedConcurrently = TRUE

      RunnableEntity InvalidateNvBlock
            symbol "NvM_InvalidateNvBlock"
            canbeInvokedConcurrently = TRUE

      RunnableEntity SetBlockProtection
            symbol "NvM_SetBlockProtection"
            canbeInvokedConcurrently = TRUE

      // for each port providing the NvMService Interface:
      PortArgument {port= PS2, value.type=BlockIdType, value.value=2}
      …
      PortArgument {port= PS<nn>, value.type=BlockIdType, value.value=<nn>}

      // for each port providing the NvMAdministration Interface:
      PortArgument {port= PAdmin<xx>, value.type=BlockIdType,
                        value.value=<xx>}
      …
      // end of section "runnable entities"
};
```

## 11.6 Configuration of the Block IDs

The Block IDs of the NvM module are modeled as “port defined argument values”. Thus the configuration of those values is part of the input to the RTE generator. Pre-compile configuration can be done by changing the XML specification for the argument values on the NVRAM Service at ECU integration time. Note that the ports visible on the client side are not affected by this.